XCS234 Assignment 1

Due Sunday, August 31 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs234-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. For SCPD classes, it is also important that students avoid opening pull requests containing their solution code on the shared assignment repositories. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing these files, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do NOT make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and can be run locally.

- `hidden`: These unit tests are NOT visible locally. These hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned. These tests will evaluate elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None   <-- This error caused the test to fail.
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0   <-- In this case, start your debugging in line 23 of grader.py.
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

Remote Execution

Basic and hidden tests are treated the same by the remote autograder, however the output of hidden tests will only appear once you upload your code to GradeScope. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

0 Introduction

In this assignment, you'll delve into the core principles of Reinforcement Learning by exploring Markov Decision Processes (MDPs) and applying policy and value iteration techniques. Through a combination of coding tasks and written analyses, you'll gain hands-on experience with implementing MDPs and evaluating policies. You'll start with a coding challenge that involves solving a classic MDP problem, which will help you understand the iterative process of improving policies. You'll also examine how different decision-making horizons can affect policy outcomes, and reflect on the implications of designing reward functions, particularly in scenarios where unintended behaviors may emerge. This assignment will solidify your understanding of these foundational RL concepts and prepare you for more advanced topics.

Deliverables:

(1) `submission.py`

(2) Written solutions to the questions in this document

1 RiverSwim MDP

Now you will implement value iteration and policy iteration for the RiverSwim environment (see picture below[1]) of (Strehl & Littman, 2008).

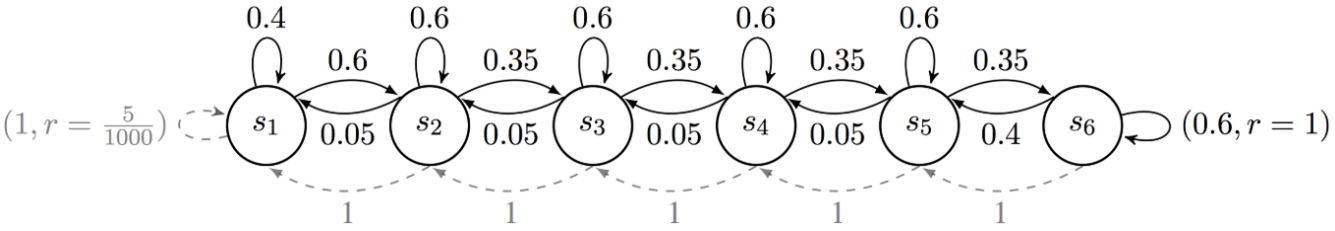

Figure 1: The RiverSwim MDP where dashed and solid arrows represent transitions for the LEFT and RIGHT actions, respectively. The assignment uses a modified, customizable version of what is shown above where there are three different strengths (WEAK, MEDIUM, or STRONG) of the current (transition probabilities for being pushed back or successfully swimming RIGHT).

Note: Run `python submission.py` for RiverSwim with a discount factor $\gamma = 0.99$ and a WEAK current, You can use this to verify your implementation. For grading purposes, we shall test your implementation against base and hidden test cases in `grader.py`.

(a) **[4 points (Coding)]**

Read through `submission.py` and implement `bellman_backup`. Return the value associated with a single Bellman backup performed for an input state-action pair.

Important: please use the version presented in the class and not any variants of it!

(b) **[4 points (Coding)]**

Implement `policy_evaluation` in `submission.py`. Return the optimal value function.

(c) **[5 points (Coding)]**

Implement `policy_improvement` in `submission.py`. Return the optimal policy.

(d) **[6 points (Coding)]**

Implement `policy_iteration` in `submission.py`. Return the optimal value function and the optimal policy.

(e) **[6 points (Coding)]**

Implement `value_iteration` in `submission.py`. Return the optimal value function and the optimal policy.

(f) **[3 points (Written)]**

Run both methods on RiverSwim with a WEAK current strength and find the largest discount factor (**only** up to two decimal places) such that an optimal agent starting in the initial far-left state (state s_1 in Figure 1) **does not** swim up the river (that is, does not go RIGHT). Using the value you find, interpret why this behavior makes sense. Now repeat this for RiverSwim with MEDIUM and STRONG currents, respectively. Describe and explain the changes in optimal values and discount factors you obtain both quantitatively and qualitatively.

The largest discount factor to now swim upward at s_1 is 0.65. It makes sense because there is a small immediate reward to swim down where it takes some effort to get the bigger reward to get to the top. Logically, it makes sense to continue moving in order to get the largest reward. If the agent is very lucky, the expected return to go straight from s_1 to the top is $(0.6 \times 0.31 \times 0.31 \times 0.31 \times 0.31 \times 0.6) \times 1 = 0.003$. It is less than the immediate reward of giving up. Therefore, it makes sense for the discount factor to be slightly larger than 0.5 to let the agent start taking risks.

[1] Figure copied from (Osband & Van Roy, 2013).

For MEDIUM and STRONG current, the largest discount factors are 0.76 and 0.92. This also makes sense because the stronger the current is, the less chance the agent can get to the top. We can do the same upper bond calculation as WEAK current. The lucky expected return for MEDIUM and STRONG is 0.0008 and 0.0001.

2 Effect of Effective Horizon

Consider an agent managing inventory for a store, which is represented as an MDP. The stock level s refers to the number of items currently in stock (between 0 and 10, inclusive). At any time, the agent has two actions: sell (decrease stock by one, if possible) or buy (increase stock by one, if possible).

- If $s > 0$ and the agent sells, it receives $+1$ reward for the sale and the stock level transitions to $s - 1$. If $s = 0$ nothing happens.
- If $s < 9$ and the agent buys, it receives no reward and the stock level transitions to $s + 1$.
- The owner of the store likes to see a fully stocked inventory at the end of the day, so the agent is rewarded with $+100$ if the stock level ever reaches the maximum level $s = 10$.
- $s = 10$ is also a terminal state and the problem ends if it is reached.

The reward function, denoted as $r(s, a, s')$, can be summarized concisely as follows:

- $r(s, \text{sell}, s - 1) = 1$ for $s > 0$ and $r(0, \text{sell}, 0) = 0$
- $r(s, \text{buy}, s + 1) = 0$ for $s < 9$ and $r(9, \text{buy}, 10) = 100$. The last condition indicates that transitioning from $s = 9$ to $s = 10$ (fully stocked) yields $+100$ reward.

The stock level is assumed to always start at $s = 3$ at the beginning of the day. We will consider how the agent's optimal policy changes as we adjust the finite horizon H of the problem. Recall that the horizon H refers to a limit on the number of time steps the agent can interact with the MDP before the episode terminates, regardless of whether it has reached a terminal state. We will explore properties of the optimal policy (the policy that achieves highest episode reward) as the horizon H changes.

Consider, for example, $H = 4$. The agent can sell for three steps, transitioning from $s = 3$ to $s = 2$ to $s = 1$ to $s = 0$ receiving rewards $+1$, $+1$, and $+1$ for each sell action. At the fourth step, the inventory is empty so it can sell or buy, receiving no reward regardless. Then the problem terminates since time has expired.

(a) **[2 points (Written)]**

Starting from the initial state $s = 3$, it possible to a choose a value of H that results in the optimal policy taking both buy and sell steps during its execution? Explain why or why not.

Yes, if H is large enough, the agent can loop between two states buying and selling stocks to gain more rewards than the full stock reward.

(b) **[2 points (Written)]**

For what values of H does the optimal policy reach a fully stocked inventory, starting from the initial state $s = 3$? I.e. Give a range for H. ***Note 1:*** *we consider the inventory fully stocked if a buy action is chosen in state $s = 9$, causing a transition to $s = 10$. This includes the last time step in the horizon.* ***Note 2:*** *By executing only buy actions, the agent can reach $s = 10$ from $s = 3$ in $H = 7$ steps.*

It is impossible to reach full stock for $H < 7$. For any $H > 200$, the agent can just buy and sell repeatedly to gain more than 100 rewards. For any value in $H \in [7, 200)$, it's impossible to reach 100 rewards by repeating buy and sell so the optimal policy is keep buying to get the full stock.

(c) **[2 points (Written)]**

Now consider the infinite-horizon discounted setting. That is, there is no time limit – the problem can only terminate when a terminal state is reached. Suppose $\gamma = 0$. What action does the optimal policy take when $s = 3$? What action does the optimal policy take when $s = 9$?

For $\gamma = 0$, the agent will only care about the instant reward. So at $s = 3$, only selling can earn reward while at $s = 9$, buying earns more than selling by reaching full stock.

(d) **[2 points (Written)]**

In the infinite-horizon discounted setting, is it possible to choose a fixed value of $\gamma \in [0, 1)$ such that the optimal policy starting from $s = 3$ never fully stocks the inventory? You do not need to propose a specific value, but simply explain your reasoning either way.

Because the MDP policy is only relevant to its current state and actions always succeed, the optimal policy will either trap in a buy-sell loop or keep buying until full stock. Even when γ close to 1, because the horizon is infinite, it is very likely that the policy will trap the agent in the buy-sell loop because that can easily yield more rewards than the full stock reward. Thus, I believe that every value in $[0, 1)$ will result in an optimal policy that never fully stock the inventory.

3 Reward Hacking

Question 2 illustrates how the particular horizon and discount factor may lead to very different policies, even with the same reward and dynamics model. This may lead to unintentional reward hacking, where the resulting policy does not match a human stakeholder's intended outcome. This problem asks you to think about an example where reward hacking may occur, introduced by Pan, Bhatia and Steinhardt[2]. Consider designing RL for autonomous cars where the goal is to have decision policies that minimize the mean commute for all drivers (those driven by humans and those driven by AI). This reward might be tricky to specify (it depends on the destination of each car, etc) but a simpler reward (called the reward "proxy") is to maximize the mean velocity of all cars. Now consider a scenario where there is a single AI car (the red car in the figure) and many cars driven by humans (the grey car).

In this setting, under this simpler "proxy" reward, the optimal policy for the red (AI) car is to park and not merge onto the highway.[3]

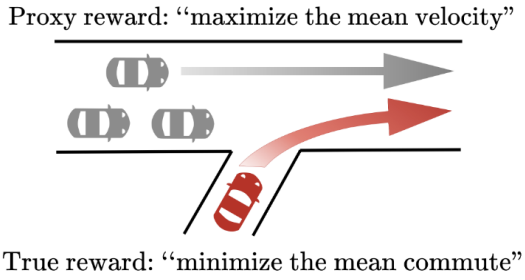

Figure 2: Pan, Bhatia, Steinhardt ICLR 2022; https://openreview.net/pdf?id=JYtwGwIL7ye

(a) **[2 points (Written)]**

Explain why the optimal policy for the AI car is not to merge onto the highway.

Because multiple gray cars need to slow down to let the red car merge. If the red car chooses not to merge, all gray cars can continue to go at high speed with a very small decrease in the mean velocity caused by the red car stopping.

(b) **[2 points (Written)]**

Note this behavior is not aligned with the true reward function. Share some ideas about alternate reward functions (that are not minimizing commute) that might still be easier to optimize, but would not result in the AI car never merging. Your answer should be 2-5 sentences and can include equations: there is not a single answer and reasonable solutions will be given full credit.

Because all cars can still reach the speed limit after merging. Mean velocity is still a good metrics to monitor, but we can add a penalty when the agent stops the car for too long, forcing it to keep moving in the long run.

[2]ICLR 2022 https://openreview.net/pdf?id=JYtwGwIL7ye

[3]Interestingly, it turns out that systems that use simpler function representations may reward hack less in this example than more complex representations. See Pan, Bhatia and Steinhardt's paper "The Effects of Reward Misspecification: Mapping and Mitigating Misaligned Models" for details.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the —README.md— for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.f

The largest discount factor to now swim upward at s_1 is 0.65. It makes sense because there is a small immediate reward to swim down where it takes some effort to get the bigger reward to get to the top. Logically, it makes sense to continue moving in order to get the largest reward. If the agent is very lucky, the expected return to go straight from s_1 to the top is $(0.6 \times 0.31 \times 0.31 \times 0.31 \times 0.31 \times 0.6) \times 1 = 0.003$. It is less than the immediate reward of giving up. Therefore, it makes sense for the discount factor to be slightly larger than 0.5 to let the agent start taking risks.

For MEDIUM and STRONG current, the largest discount factors are 0.76 and 0.92. This also makes sense because the stronger the current is, the less chance the agent can get to the top. We can do the same upper bond calculation as WEAK current. The lucky expected return for MEDIUM and STRONG is 0.0008 and 0.0001.

2.a

Yes, if H is large enough, the agent can loop between two states buying and selling stocks to gain more rewards than the full stock reward.

2.b

It is impossible to reach full stock for $H < 7$. For any $H > 200$, the agent can just buy and sell repeatedly to gain more than 100 rewards. For any value in $H \in [7, 200)$, it's impossible to reach 100 rewards by repeating buy and sell so the optimal policy is keep buying to get the full stock.

2.c

For $\gamma = 0$, the agent will only care about the instant reward. So at $s = 3$, only selling can earn reward while at $s = 9$, buying earns more than selling by reaching full stock.

2.d

Because the MDP policy is only relevant to its current state and actions always succeed, the optimal policy will either trap in a buy-sell loop or keep buying until full stock. Even when γ close to 1, because the horizon is infinite, it is very likely that the policy will trap the agent in the buy-sell loop because that can easily yield more rewards than the full stock reward. Thus, I believe that every value in $[0, 1)$ will result in an optimal policy that never fully stock the inventory.

3.a

Because multiple gray cars need to slow down to let the red car merge. If the red car chooses not to merge, all gray cars can continue to go at high speed with a very small decrease in the mean velocity caused by the red car stopping.

3.b

Because all cars can still reach the speed limit after merging. Mean velocity is still a good metrics to monitor, but we can add a penalty when the agent stops the car for too long, forcing it to keep moving in the long run.