Assignment #3- MDP & Learning

Read all the instructions before starting

General Instructions

- The assignment is due to 26/01/2023, 23:59.
- The assignment should be submitted in pairs and in pairs only.
- The answers shall be typed – handwritten solutions will not be accepted.
- Questions shall be asked only via Piazza
- The TA in charge of this assignment is Or Raphael Bidusa
- Justified late submissions requests should be sent only to the teaching assistant in charge (Sapir Tubul)
- During the exercise there may be updates, a corresponding announcement will be sent.
- This assignment worth is 10% of your final grade and therefore copying and other forms of cheating will be severely dealt with.
- You should include the answers to the questions with the ✍ mark next to them in your report.
- For the wet part, you are given the general template of the code.
- We appreciate and listen to your complaints about the assignment and update this document accordingly. Updated versions of this document will be upload to the course's site. Updates and clarifications added after the first version will be marked in yellow. There may be many versions – don't panic! The changes might be small and barely significant.

Make sure that you are using only the python libraries permitted in each wet part.
Code with additional libraries will not be accepted.

It is recommended to go through the lectures and tutorials once again before starting.

Part A – MDP & RL (51 pt.)

Background

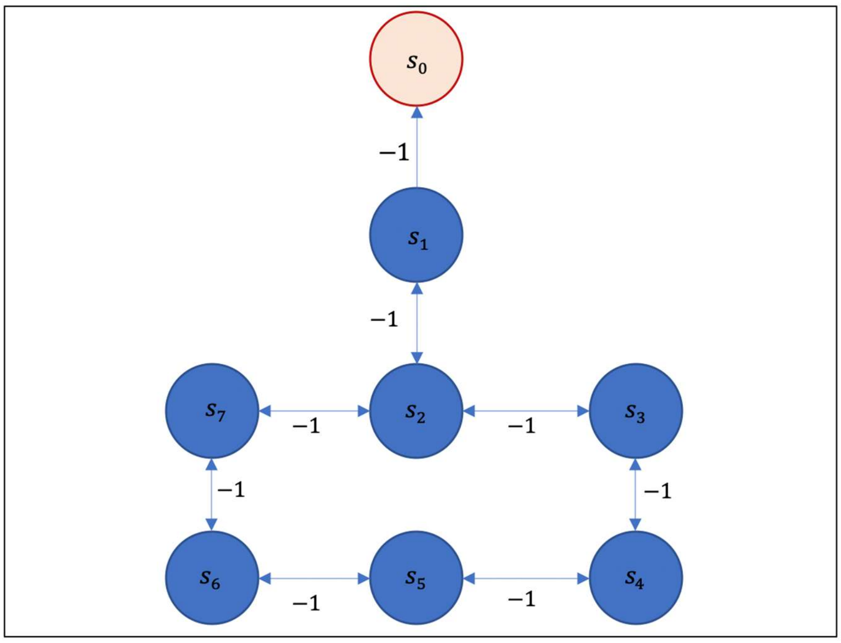
In this part we shall work with Markov Decision Processes, with infinite horizon (stationary policy).

Part A – dry part ✍

1. In the tutorial we've seen Bellman's equations when the reward function was a function of the current state, meaning $R: S \to \mathbb{R}$. This reward function is called "State-Reward" because it is depended on the given state and the given state only.
   Now, we shall expand this idea, for a reward as a function of the current state, the chosen action and the next state that was (non-deterministically) chosen, meaning $R: S \times A \times S' \to \mathbb{R}$. This reward function is called "Edge-Reward".
   a) (2 pt.) Change the formula for the expected utility from the tutorial for the "Edge-Reward" case, no explanation needed.
   b) (2 pt.) Rewrite Bellman's equation for the "Edge-Reward" case, no explanation needed.
   c) (4 pt.) Rewrite the "Value Iteration" pseudo-code for the "Edge-Reward" case.
   d) (4 pt.) Rewrite the "Policy Iteration" pseudo-code for the "Edge-Reward" case.

   For subsections (c) & (d), make sure to explain the case when $\gamma = 1$ and explain, according to your opinion, what are the needed conditions on the MDP\environment for successfully finding the optimal policy.

The following graph is given:



And the following:

- (Discount factor) $\gamma = 1$.
- Infinite horizon.
- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ – set of states – describing the position of the agent in the graph.
- $S_G = \{s_0\}$ – set of terminal states.
- The action set for each state in S, for example $A(s_2) = \{\uparrow, \rightarrow, \leftarrow\}$.
- "Edge-Reward" function:
$$\forall s \in S \backslash S_G, a \in A(s), s' \in S: \;\; R(s, a, s') = -1$$
- Deterministic transition function – each action succeeds with the probability of 1.

e) (Dry, 6 pt.) Run the Value Iteration algorithm you rewrote in subsection (c) on the given graph. Fill in the values in the following table when $\forall s \in S: U_0(s) = 0$. (You may not need to fill it all out)

	$\boldsymbol{U_0(s_i)}$	$\boldsymbol{U_1(s_i)}$	$\boldsymbol{U_2(s_i)}$	$\boldsymbol{U_3(s_i)}$	$\boldsymbol{U_4(s_i)}$	$\boldsymbol{U_5(s_i)}$	$\boldsymbol{U_6(s_i)}$	$\boldsymbol{U_7(s_i)}$	$\boldsymbol{U_8(s_i)}$
s_1	0								
s_2	0								
s_3	0								
s_4	0								
s_5	0								
s_6	0								
s_7	0								

f) (Dry, 6 pt.) Run the Policy Iteration algorithm you rewrote in subsection (d) on the given graph. Fill in the values in the following table when the initial policy is already given in the first column. (You may not need to fill it all out)

	$\boldsymbol{\pi_0(s_i)}$	$\boldsymbol{\pi_1(s_i)}$	$\boldsymbol{\pi_2(s_i)}$	$\boldsymbol{\pi_3(s_i)}$	$\boldsymbol{\pi_4(s_i)}$	$\boldsymbol{\pi_5(s_i)}$	$\boldsymbol{\pi_6(s_i)}$	$\boldsymbol{\pi_7(s_i)}$	$\boldsymbol{\pi_8(s_i)}$
s_1	$\uparrow$								
s_2	$\uparrow$								
s_3	$\leftarrow$								
s_4	$\uparrow$								
s_5	$\rightarrow$								
s_6	$\rightarrow$								
s_7	$\downarrow$								

Part B – Getting comfortable with the code

This part of the assignment is only for getting to know the code better, read it all and make sure that you fully understand the entire code and structure.

mdp.py – you don't need to alter this file at all.

In this file the MDP environment is implemented, the constructor gets the following parameters:

- board – defines the possible states in the MDP and the reward for each state, AKA "State-Reward" case.
- terminal_states - the set of all terminal states (must have at least one).
- transition_function – the transition function – for a chosen action gives the probability of each action to actually be executed.
- Discount factor – gamma, gets values of $\gamma \in (0,1)$.
  In this assignment we will not check the case of $\gamma = 1$.

Note: the action set is defined in the constructor itself and is the same for all board that may be chosen.

The MDP class has some functions that you may find useful throughout the next part.

- print_rewards() – prints the board with the reward for each state.
- print_utility(U) – prints the board with the utility value U for each state.
- print_policy(policy) – prints the board with the action chosen by "policy" for each non-terminal state.
- step (state, action) – given the current "state" and "action", returns the next state deterministically. Trying to walk through a "wall-state" or outside the board will keep the agent in place – thus returning the same given "state".

Part C – wet part

The entire code should be written in the mdp_rl_implementation.py file.

The following libraries are allowed to be used:

All the built-in packages in python, numpy, matplotlib, argparse, os, copy, typing, termcolor, random

You need to implement the following methods:

- (7 pt.) value_iteration(mdp, U_init, epsilon) – given an MDP, initial utility value U_init and an upper bound for the optimal utility error - epsilon, runs the Value Iteration algorithm and returns the U resulting in the end of the run. **TODO**

- (7 pt.) get_policy(mdp, U) – given an MDP and utility value U (which satisfies the Bellman equation) returns the corresponding policy (In case there are several corresponding policies, returns one of them). **TODO**

- (7 pt.) q_learning(mdp, init_state,…) – given the MDP, initial state init_state and the other parameters needed for the algorithm, runs the Qlearning algoritm and returns the Qtable resulting in the end of the run. **TODO**

  NOTE! Rembember that the Qlearning algorithm is ActiveRL-modelfree one and therefore should not have received the MDP as a parameter but an environment simulator instead.

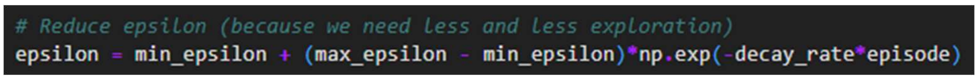
  The transition function of the environment and the rewards are given from the said environment as a result of simulating a run.

  - Start with a Qtable with all zeros.
  - As you remember from the lecture, updating the value of a cell in the Qtable is done according to the following equation:

    $$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$$

    In this function α is the "learning_rate" parameter given to the method.

    - As you remember from the lecture, this algorithm requires simulating runs on the environment – each simulation is called “an episode”.
      Each simulation will start form the init_state (given as a parameter) following by a sequence of actions that ends when a terminal state is met or after max_steps number of steps - whichever occurs sooner.
      During the simulation, given a state s we need to chose an action according to an decision rule, for this methods you will implement the “$\varepsilon - greedy$" rule:
      Every time an action must be chosen, we will randomly draw a sample from a uniform distribution.
      If the sampled value is greater than ε then the action which maximized the Qtable value for s is chosen (in case of multiple max values – choose an arbitrary one).
      Otherwise, choose a random action uniformly over the set of all actions.
      At the end of each episode (and not at the end of each step) update the value of ε according to the following line:



```
# Reduce epsilon (because we need less and less exploration)
epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
```

      When “episode” is the number of the current episode (starting from 0).
      You are more than welcome to use and copy the given code.
      This is the only place that you should use the “decay_rate”, “min\max_epsilon” parameters.

- (3 pt.) q_table_policy_extraction(mdp,qtable) – given the MDP and the Qtable, returns the corresponding policy. If there are multiple corresponding policies, choose one arbitrarily.

- ✍ (Dry, 3 pt.) John Doe ran the Qlearning algorithm on an MDP with a positive reward for each state and an initial Qtable with all zeros.

  At the end of the run he printed the table and saw that some of the values of certain states are still 0 for certain actions. Explain how such situation can occur.

You can find example for using all the methods above in the main.py file.
In the beginning of the main function the environment is loaded from these three files: board, terminal_states, transition_function
And by thus creating an instance of the environment (the MDP).

- Note that the current code in the main function cannot run because you need to implement the relevant methos in the mdp_rl_implementation.py file.
- Additionally, in order to print the board with colors you need to run the code in an IDE, like PyCharm.

The next subsections are bonus – not mandatory (5 pt.)

In order to get the bonus the BOTH METHODS should be implemented, otherwise – keep them unimplemented.

- policy_evaluation(mdp, policy) – Given the mdp and a policy, returns the utility values corresponding for each state. **TODO**

- policy_iteration(mdp, policy_init)- Given the mdp and an initial policy, runs the policy iteration algorithm and returns the optimal policy. **TODO**

Part B – Intro to Learning (49 pt.)

✍ Part A – dry part (7 pt.)

Questions 1 & 2 are no longer mandatory!

No points will be given on them, but you are still welcome to submit them for feedback.

1. (No Points) Let $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$ be a dataset from the size of n and a binary labeling, $y_i \in \{0,1\}$.
Each example in the dataset is a vector of two continues features $x_i = (v_1, v_2)$.
Assume that there exists a classifier $f(x): R^2 \to \{0,1\}$ which we want to learn (it is not given to us) and assume that the examples are consistent with the classifier (there is no noise in the dataset).
In the following subsections, for KNN consider an Euclidean distance function.
Also, assume that if, when classifying a point, there are several examples at the same distance, first consider examples with a maximal v_1 value and in case of equality in v_1 values, first consider examples with a maximal v_2 value.
Assume that there are no identical examples in the dataset (with both the same v_1 value and the same v_2 value).
In the KNN algorithm, a points is not a neighbor of itself.
In each subsection **show an example for such dataset that satisfies all the conditions, write an explanation and add a graphical representation that shows such case (including at least the classifier and your chosen dataset).** Mark positive labeling with a '+' sign and negative labeling with a '-' sign. In each following subsection you are not allowed to choose a trivial classifier – meaning a classifier that gives all examples a positive labeling or gives all examples a negative labeling.

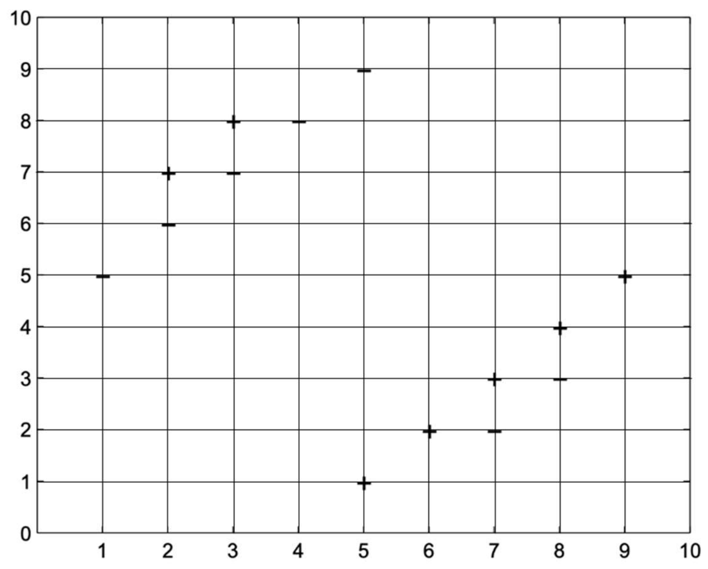
   a) Give an example of a classifier $f(x): R^2 \to \{0,1\}$ and a training set with no more than 10 examples such that learning a decision tree with ID3 will yield a classifier which labels any point in R^2 correctly but learning with KNN will yield a classifier which labels at least one point in R^2 wrong.

   b) Give an example of a classifier $f(x): R^2 \to \{0,1\}$ and a training set with no more than 10 examples such that learning with KNN will yield a classifier which labels any point in R^2 correctly but learning a decision tree with ID3 will yield a classifier which labels at least one point in R^2 wrong.

c) Give an example of a classifier $f(x): R^2 \rightarrow \{0,1\}$ and a training set with no more than 10 examples such that learning both with KNN and a decision tree with ID3 will yield a classifier which labels at least one point in R^2 wrong.

d) Give an example of a classifier $f(x): R^2 \rightarrow \{0,1\}$ and a training set with no more than 10 examples such that learning both with KNN and a decision tree with ID3 will yield a classifier which labels any point in R^2 correctly.

2. (No Points) In this question we will use a k-nearest neighbour using Euclidean distance function, and a binary labeling.
We shall define the classification of a point to be the classification of the majority of it's k nearest neighbors (NOTE: in this specific question a point can be a neighbor of itself).
In case of draw the chosen classification will be "True" (positive).



a) What value of k minimizes the training error for the given training set? What is the training error resulting from choosing such value? Draw the decision line of the KNN resulting from choosing such value.
b) Explain why using too low or too high values of k is bad for this specific training set.
c) Read about Leave-One-Out Cross Validation in the following link: https://www.statology.org/leave-one-out-cross-validation/
What values of k minimize the Leave-One-Out Cross Validation error for the given training set? What is that error?

This question is still mandatory :)

3. (7 pt.) Consider a decision tree T, a test example $x \in \mathbb{R}^d$ and a vector $\varepsilon \in \mathbb{R}^d$ s.t $\forall i \in [1, d]: \varepsilon_i > 0$.
An epsilon-decision rule differs from the normal decision rule learnt in class in such way:
Consider arriving to a node in the decision tree that splits the tree according to the i_{th} feature with the threshold value of v_i while trying to classify x.
If $|x_i - v_i| \leq \varepsilon_i$ then the classification process continues on both children of the node instead of just one.
In the end, x is classified according to the most common label of all of the examples in all of the leaves reached in the classification process. (In case of a draw the classification will be "True").

Let T be a decision tree and T' be the tree resulting by pruning the last level of T (meaning all the examples belong to each pair of sibling leaves were passed to the parent).

Prove or disprove: There exists a vector ε such that T with an epsilon-decision rule and T' with the normal decision rule will classified every testing example in $\mathbb{R}^d$ identically.

Part B – Getting comfortable with the code

Background

This part of the assignment is only for getting to know the code better, read it all and make sure that you fully understand the entire code and structure.
In the part of learning we shall use a dataset. The data is split into two sets: the training set train.csv and the test set test.csv.
Generally speaking, the training set will help us learn and build the classifiers and the test set will help us evaluate their performance.

In the utils.py file you can find the following methods for your own use:
load_data_set, create_train_validation_split, get_dataset_split
Which loads\splits the data from the csv files to np.array (read the methods' documentation).

The data for the ID3 in this assignment includes features collected from scans that meant to help us differentiate between benign and malignant tumors. Each example consists of 30 features and a binary column under the name of diagnosis that determines the type of the tumor (0=benign, 1=malignant). All the features are continuous. The first column states if the patient is ill (M) or healthy (B). The other columns state other medical indices of the patient (the indices are complicated, and you don't need to understand their meaning at all).

$\underline{ID3 - dataset}$ folder:

- Includes the data csv files for the ID3.

$\underline{utils.py}$ file:

- Includes auxiliary methods that will be useful throughout the code, like for loading a dataset and calculating the accuracy.
- In the next part you will implement the "l2_dist" and "accuracy" methods. Read the methods' documentation and the comments next to the description of each method **TODO**.

$\underline{unit\_test.py}$ file:

- Basic testing file that can help you test you implementation.

$\underline{DecisionTree.py}$ file:

- This file includes 3 useful classes for building out ID3 tree:
    - $\underline{Question}$: This class implements the branching of a node in a tree. It saves the feature and the threshold value to split the data with.
    - $\underline{DecisionNode}$: This class implements a node in the decision tree.
      The node includes a $Question$ and the two children $true\_branch$ and $false\_branch$.
      $true\_branch$ is the branch for the examples which answer $True$ on the $Question$ (the $match$ function of the $Question$ returns $True$).

While the $false\_branch$ is the branch for the examples which answer $False$ on the $Question$ (the $match$ function of the $Question$ returns $False$).

   - $\underline{Leaf:}$ This class implements a leaf in the decision tree. For each label in the dataset the leaf includes the number of examples with this label, for example $\{'B': 5, M': 6\}$.

$\underline{ID3.py}$

- This file includes the $ID3$ class that you need to implement.
  Read the documentation and the methods' description.

$\underline{ID3\_experiments.py}$:

- This file includes the methods needed to run the experiments and testing on the ID3, the file includes the following experiments that will be explained later:

$$cross\_validation\_experiment, basic\_experiment$$

Part C – wet part (42 pt.)

For this part the following libraires are permitted:

All the built in packages in python, sklearn, pandas ,numpy, random, matplotlib, argparse, abc, typing.

<u>But of course you cannot use any learning algorithm or other data structure that is part of the learning algorithm that you will be asked to implement.</u>

4. (5 pt.) complete the utils.py file by implementing the $l2\_dist$ and $accuracy$ methods. Read the documentations and the descriptions of these methods. **TODO**
(Run the corresponding tests in the $unit\_test.py$ file to make sure your implementation is valid).
Note! In the documentation there are restrictions on the code itself, not following them will result in points being deducted.
Additionally, change the ID value in the beginning of the file from 123456789 to one of the submitters' ID.

5. (25 pt.) ID3 Algorithm:

a. Complete the ID3.py file by implementing the ID3 algorithm as shown in class. **TODO**
Note that all of the features are continuous and therefore you are asked to use the method of dynamic auto-discretization shown in the lecture. When examine a value for splitting a continuous feature, examples with feature value that equals to the threshold value will be treated as if they had a value higher than the threshold value. In case of multiple optimal features to split with – choose the feature with the minimal index out of them.

✍ **b.** Implement the $basic\_experiment$ method in $ID3\_experiments.py$ **TODO**
Run the corresponding part in $main$ and add the accuracy in your report.

6. (8 pt.) Pre-pruning.
Splitting a node in the tree happens as long as the number of its examples is larger than the minimum bound m, causing a "pre-pruning" effect as learnt in class.
Note that in that case the resulting tree might not be consistent with the training set. After the learning process (of a single tree), the classification of a new a new example is determined by the classification of the majority of examples in the corresponding leaf.

✍ **a.** (3 pt.) Explain the importance of the pruning in general and state what phenomena it tries to prevent.

b. (5 pt.) Update your implementation in the $ID3.py$ file such that pre-pruning will occur as learnt in class. The parameter $min\_for\_pruning$ states the minimal number of examples in a node for it to be a leaf, meaning the pruning shall occur if and only if the number of examples in the node is smaller or equal to that parameter. **TODO**

c. This subsection is bonus (5 pt)

Note! This is a dry question and there is no need to submit the code you wrote for it.

1. Choose at least 5 different values for the parameter.
2. For each value, calculate the accuracy of the classifier using $\mathrm{K-}$ $\mathbf{fold\ cross\ validation}$ on the training set and the training set only.

For splitting the training set into K subsets use the function sklearn.model_selection.KFold with the parameters of $\mathrm{n\_split} = 5$,$\mathrm{shuffle} = \mathrm{True}$ and $random\_state$ equal to one of the submitters' ID.

✍ i. Use the results and make a graph of the accuracy as a function of M.
Add the graph to your report. (You can use the method $util\_plot\_graph$ in the $utils.py$ file).

✍ ii. Explain the graph. Which pruning got the best accuracy? What was the accuracy?

The bonus section is over, the next section is mandatory.

✍ **d.** (4 pt.) Use the ID3 algorithm with the pre-pruning in order to build a classifier from the entire training set and make a prediction on the test set.
Use the best M value found in subsection (c). (Implement $best\_m\_test$ method in the $ID3\_experiments.py$ file and run the corresponding part in $main$).
Add to your report the accuracy for the test set. Did the pruning improved the preferments compared to the un-pruned classifier in section 5?
In this subsection, if you did not implement subsection c, use the value $M = 50$.

Submission Instructions

- ✓ The assignment shall be submitted online in pairs only.
- ✓ Your code will be also checked automatically so be sure to follow the requested format. A submission that does not comply with the format will not be checked (score of 0).
- ✓ Making up data for the purpose of building the graphs is prohibited and constitutes a disciplinary offense.
- ✓ Keep your code clean and documented. The answers in your report should be according to the order of the corresponding questions.
- ✓ Submit a single zip in the name of AI3_<id1>_<id2>.zip (without the angle brackets) including:

- Your report – a file with the name of AI_HW3.PDF including your answers to the dry questions.

- The code files that you were required to implement in the exercise and those only:
    - utils.py
    - For the part of the decision trees – ID3.py, ID3_experiments.py
    - For the part of the MDP & RL- mdp_rl_implementation.py