

# Assignment #3- MDP & Learning

Read all the instructions before starting

## General Instructions

- The assignment is due to 06/07/2023, 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 15% 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 (60 pt.)

### Background

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

### Part A – dry part 🍌

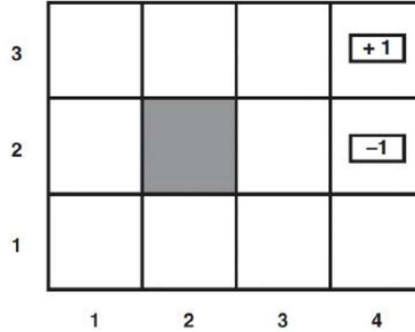
1. In the tutorial we've seen Bellman's equations when the reward function was a function of the current state, meaning  $R: S \rightarrow \mathbb{R}$ . This reward function is called "State-Reward" because it is depended on the given state and the given state only.

Now, we shall expand this idea, for a reward as a function of the current state and the chosen action, meaning  $R: S \times A \rightarrow \mathbb{R}$ . This reward function is called "Action-Reward".

- a) (2 pt.) Change the formula for the expected utility from the tutorial for the "Action-Reward" case, no explanation needed.
- b) (2 pt.) Rewrite Bellman's equation for the "Action-Reward" case, no explanation needed.
- c) (4 pt.) Rewrite the "Value Iteration" pseudo-code for the "Action-Reward" case.
- d) (4 pt.) Rewrite the "Policy Iteration" pseudo-code for the "Action-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.

2. The following MDP is given,  $\langle S, A, P, R, \gamma \rangle$ , infinite horizon:



States:

$$S = \{(1,1), (1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,3), (3,4)\}$$

$$S_G = \{(2,4), (3,4)\}$$

Actions:

$$\forall S \setminus S_G: A(s) = \{Up, Down, Left, Right\}$$

Rewards:

The only rewards given are those of the terminal states:  $R((2,4)) = -1, R((3,4)) = +1$ .

Notice that the function is a “State-Reward” function.

There are reward values for the other states, but they are not given as part of the question.

Transition Function:

Every action “succeeds” in a probability of 0.8, and if it does not succeed, then with equal probability one of the actions perpendicular to the requested action is performed. When an agent goes towards a wall or off the board it stays in place.

Discount Factor:  $0 < \gamma < 1$ .

You ran the *value iteration* algorithm with  $\varepsilon \rightarrow 0$  and got the following output:

( $\varepsilon \rightarrow 0$  means that the stopping condition fulfill the condition that the infinity norm between the utility vectors was zero, that is, the utility values obtained from the algorithm satisfy the Bellman equation).

3	$v_4$	$v_6$	$v_9$	<span style="border: 1px solid black; padding: 2px;">+1</span>
2	$v_2$		$v_7$	<span style="border: 1px solid black; padding: 2px;">-1</span>
1	$v_1$	$v_3$	$v_5$	$v_8$
	1	2	3	4

When  $v_i$  is the utility value for the  $i$ th state as shown in the graph. Additionally, we will use  $r_i$  as the reward value for the  $i$ th state.

Answer true / false and provide a brief explanation or a detailed counterexample.

- a) (3 pt.) If  $v_9 > 1$  then necessarily  $r_9 > 1$ . True/False.

Explanation / A Counterexample:

- b) (3 pt.) If  $\forall i \in [9]: v_i > 0$  then necessarily  $\exists i \in [9]: r_i > 0$ . True/False.

Explanation / A Counterexample:

- c) (3 pt.) If  $r_1 = r_2 = \dots = r_9 < 0$  then necessarily  $v_1 = \min\{v_i | i \in [9]\}$ . True/False.

Explanation / A Counterexample:

- d) (3 pt.) If  $v_1 > v_2 > v_3 > 0$  then necessarily  $\pi^*((1,1)) = Up$ . True/False.

Explanation / A Counterexample:

- e) (2 pt.) If  $\gamma = 0$ , what is the number of optimal policies? Explain.

- f) (2 pt.) If  $\gamma = 1, v_5 = -1$ , what is  $\pi^*((1,4))$  as a function of  $r_8$ ? Explain.

- g) (2 pt.) It is given that  $v_1 > v_2 > v_3 > 0$ . Find a tight upper and lower bounds for  $r_1$  as a function of  $v_i$  (and not as a function of  $\gamma$ ).

## 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_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:

- (10 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**
- (10 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**
- (5 pt.) `policy_evaluation(mdp, policy)` – Given the `mdp` and a policy, returns the utility values corresponding for each state. **TODO**
- (10 pt.) `policy_iteration(mdp, policy_init)` - Given the `mdp` and an initial policy, runs the policy iteration algorithm and returns the optimal policy. **TODO**

**For terminal states and WALL, the value in the policy table should be is None. Any other value will not be accepted as an answer.**

**For WALL the value in the value table should be None. Any other value will not be accepted as an answer.**

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 methods in the `mdp_implementation.py` file.
- Additionally, in order to print the board with colors you need to run the code in an IDE, like PyCharm.

## Part B – Intro to Learning (40 pt.)

### 👉 Part A – dry part (20 pt.)

#### kNN - getting to know it

In this section you will get to know a nifty learning algorithm called kNN, or in its full name k-Nearest Neighbors, when  $k$  is actually a parameter of the algorithm!

Let  $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$  be a training set from the size of  $n$  when  $\forall i: x_i \in \mathbb{R}^d, y_i \in \mathcal{Y}$ . All the samples in the training set are  $d$ -dimensional vectors and the labels are from a certain domain. This is a classification problem. The classification will be binary, meaning  $\mathcal{Y} = \{-, +\}$  unless stated otherwise.

For each sample in the training set, we can see the  $i$ th entry in the vector as the value for the  $i$ th feature of the sample, meaning each sample  $x_i$  can be represented with  $d$  values

$f_1(x_i), f_2(x_i), \dots, f_d(x_i)$ .

The “training” phase of the algorithm is quite simple - simply saving the training set in its entirety. The classification phase is also quite simple - when you want to classify an example from the test set, you look at its  $k$  nearest neighbors in the  $d$ -dimensional plane among the examples in the training set and classify the example according to the most common classification among the  $k$  neighbors.

In order to avoid a tie between the classifications, we will usually assume that  $k$  is odd, or we will well define a tie breaker.

If not stated otherwise, in case of tie in a binary classification, we will classify the example as positive  $+$ .

#### Roger That?

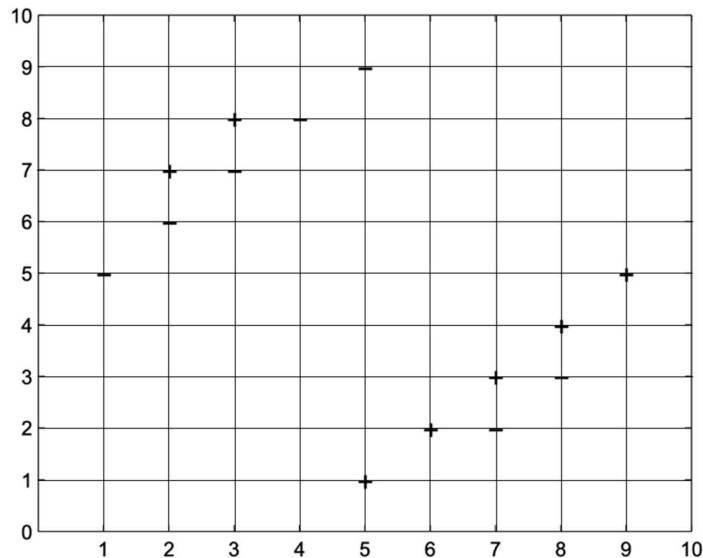
- a) (3 pt.) As mentioned before, in the classification phase we choose the most common label among the nearest neighbors, but we need to choose which distance function to define this exact set of neighbors. Two common distance functions to use are Euclidian and Manhattan.

For a binary classification problem, give a simple example for  $k, d$ , a training set and a test sample in which using a different distance function (among these two) yields a different classification of the test sample.

For the next subsections we will use the Euclidian distance function, unless stated otherwise.



The following training set is given, when  $d = 2$ :



- (1 pt.) What value of  $k$  minimizes the training error for the given training set? What is the training error resulting from choosing such value?
- (1 pt.) What value of  $k$  gives us a *majority* classifier? (In which each test sample gets the most common label among the entire training set).
- (2 pt.) Explain why using too low or too high values of  $k$  is bad for this specific training set.

Another variation of the  $kNN$  learning algorithm has a parameter  $r$  – a radius, instead of  $k$ . In this case, the classification of a test sample will be chosen as the majority classification of samples in the training set which are within a radius  $r$  from the test sample. In case of a tie, even if vacuously, the classification will be positive.

For simplicity, in the following subsection we will neglect the scenario in which the group of  $k$  nearest neighbors is not well defined, i.e. a situation where there are more than  $k$  nearest neighbors due to equality in distance to the test sample.

Prove or Disprove by Counterexample:

- (3 pt.)  $\exists d, k$ , a training set and a test example such that  $\nexists r$  for which the classification of the test sample in the new variation will be the same as the classification in the original version of the algorithm.
- (3 pt.)  $\exists d, r$ , a training set and a test example such that  $\nexists k$  for which the classification of the test sample in the original version of the algorithm will be the same as the classification in the new variation.

## Splitting the Fun

(7 pt.) As you already know, when classifying a sample test with a decision tree, when we encounter a node, we decide which children to pass the sample to by a threshold value  $v$  which is compared to one of the feature values of the sample. Sometimes the threshold value is really close to the feature value. We would like to take into account values which are "close" to the threshold when classifying a test sample, and not seal the sample's fate to only one subtree; For this purpose we present the following algorithm:

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  $\varepsilon$  such that  $T$  with an epsilon-decision rule and  $T'$  with the normal decision rule will classify 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).

#### ID3 – dataset folder:

- Includes the data csv files for the ID3.

#### 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 "accuracy" method. Read the method's documentation and the comments next to the description of the method **TODO**.

#### unit\_test.py file:

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

#### DecisionTree.py file:

- This file includes 3 useful classes for building out ID3 tree:
  - 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.
  - 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*).

- 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  $\{B':5, M':6\}$ .

#### ID3.py

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

#### 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 (20 pt.)


For this part the following libraires are permitted:

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

**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.**

1. (3 pt.) complete the `utils.py` file by implementing the *accuracy* method.  
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.
2. (10 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.
3. **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. (2 pt.) Explain the importance of the pruning in general and state what phenomena it tries to prevent.
  - b. (3 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  $K$  – **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 `n_split = 5`, `shuffle = 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. (2 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_implementation.py`

**The submission should not contain any directories, a submission that does not comply with the format will not be reviewed.**