Uppsala University

Machine Learning

Assignment 8

General information

- The recommended tool in this course is R (with the IDE R-Studio). You can download R **here** and R-Studio **here**. You can use Python and Jupyter Notebooks, although the assignments may use data available only through the R package, a problem you would need to solve yourself.
- Report all results in a single, `*.pdf`-file. *Other formats, such as Word, Rmd, Jupyter Notebook, or similar, will automatically be failed.* Although, you are allowed first to knit your document to Word or HTML and then print the assignment as a PDF from Word or HTML if you find it difficult to get TeX to work.
- The report should be written in English.
- If a question is unclear in the assignment. Write down how you interpret the question and formulate your answer.
- You should submit the report to **Studium**. Deadlines for all assignments are **Sunday 23.59**. See **Studium** for dates. Assignments will be graded within 10 working days from the assignment deadline.
- To pass the assignments, *you should answer all questions not marked with **, and get at least 75% correct.
- To get VG on the assignment, *also the questions marked with a ** should be answered. Sometimes you can choose between different VG assignments, then this is explicitly stated. To get VG you need 75% both on questions to pass and the VG part of the assignment. VG will only be awarded on the first deadline of the assignment.
- A report that does not contain the general information (see the **template**), will be automatically rejected.
- When working with R, we recommend writing the reports using R markdown and the provided **R markdown template**. The template includes the formatting instructions and how to include code and figures.
- Instead of R markdown, you can use other software to make the pdf report, but you should use the same instructions for formatting. These instructions are also available in **the PDF produced from the R markdown template**.
- The course has its own R package `uuml` with data and functionality to simplify coding. To install the package just run the following:

  1. `install.packages("remotes")`
  2. `remotes::install_github("MansMeg/IntroML", subdir = "rpackage")`

- We collect common questions regarding installation and technical problems in a course Frequently Asked Questions (FAQ). This can be found **here**.
- You are not allowed to show your assignments (text or code) to anyone. Only discuss the assignments with your fellow students. The student that show their assignment to anyone else could also be considered to cheat. Similarly, on zoom labs, only screen share when you are in a separate zoom room with teaching assistants.

- The computer labs are for asking all types of questions. Do not hesitate to ask! The purpose of the computer labs are to improve your learning. We will hence focus on more computer labs and less on assignment feedback. *Warning!* There might be bugs in the assignments! Hence, it is important to ask questions early on so you dont waste time of unintentional bugs.
- If you have any suggestions or improvements to the course material, please post in the course chat feedback channel, create an issue **here**, or submit a pull request to the public repository.

Contents

1 Bandits

In the first part, we will study the k-armed bandit problems more in detail.

1. Implement a stationary 5-armed bandit reward function that generates a reward R for a given category according to a true underlying (stationary) reward. Implement a function `stationary_bandit(a)` that takes an action $\mathcal{A} = (1, ..., 5)$. The reward function should return a reward according to

$$R \sim N(q_a^\star, 1)\,,$$

where $q^\star = (1.62, 1.20, 0.70, 0.72, 2.03)$. Below is an example of how it *could* work. Note that since the function returns a random value, you might get a different result, and it might still be a correct implementation. However, to test, you can run your bandit 1000 times and take the mean (see below). Then you should be close to $q^\star$ if your implementation is correct.

```
set.seed(4711)
stationary_bandit(2)

## [1] 3.019735

stationary_bandit(1)

## [1] 2.99044

R <- numeric(1000)
for(i in 1:1000){
  R[i] <- stationary_bandit(3)
}
mean(R)

## [1] 0.7099425
```

2. Also, we will also implement a non-stationary reward function. For all actions $\mathcal{A} = (2, ..., 5)$ the non-stationary should work as the stationary reward function. Although for action $a = 1$, the function should return a reward based on the iteration according to

$$R \sim N(-3 + 0.01t, 1)\,.$$

Implement the reward function as `nonstationary_bandit(a, t)` and it should work as follows. Again, note that these are random values, you might have another (correct) solution giving different results.

```
set.seed(4711)
nonstationary_bandit(2, t = 1)

## [1] 3.019735
```

```r
nonstationary_bandit(2, t = 1000)
```

```
## [1] 2.57044
```

```r
nonstationary_bandit(1, t = 1)
```

```
## [1] -1.793682
```

```r
nonstationary_bandit(1, t = 1000)
```

```
## [1] 6.593121
```

3. Implement a greedy policy algorithm that always chooses the greedy action, according to Eq. (2.2) in Sutton and Barto (2020). The function called `greedy_algorithm(Q1, bandit)` should input a vector of initial estimates Q_1 and a reward function. Then the algorithm should run the bandit for 1000 steps and return the following; (1) The rewards R_t for all steps, (2) the mean reward over all the 1000 steps, (3) the value estimates $Q_t(a)$, and (4) the total number of choices of each action $N_t(a)$. Here is an example of how it could work. *Note!* This is a random algorithm so you might get a different result, even if you use the exact same seed and have a correct algorithm.

```r
Q1 <- rep(0, 5)
set.seed(4711)
greedy_algorithm(Q1, stationary_bandit)
## $Qt
## [1] 1.593045 0.000000 0.000000 0.000000 0.000000
##
## $Nt
## [1] 1000    0    0    0    0
##
## $R_bar
## [1] 1.593045
##
## $Rt
## [1]  1.772517635  2.816318235  0.633508626  0.111087954 ...
```

4. Similarly, implement the ϵ-bandit algorithm that takes an additional argument ϵ that is the probability that the algorithm instead takes makes an explorative action. Implement this algorithm as `epsilon_algorithm(Q1, bandit, epsilon)`, note that it should return the same output structure as the greedy algorithm. It is also clear that the randomness of the exploration can give quite some different results in different runs.

```r
set.seed(4711)
Q1 <- rep(0, 5)
epsilon_algorithm(Q1, stationary_bandit, 0.1)
## $Qt
## [1] 1.6093183 1.1491153 0.7382012 0.5975228 2.0272784
##
```

```
## $Nt
## [1] 234  17  19  16 714
##
## $R_bar
## [1] 1.867178
##
## $Rt
## [1]  1.77251764  2.81631824  0.63350863  2.01754941 ...

set.seed(4712)
Q1 <- rep(0, 5)
epsilon_algorithm(Q1, stationary_bandit, 0.1)
## $Qt
## [1] 1.2660095 1.4034524 0.4066058 0.4252434 1.9752285
##
## $Nt
## [1]  21  16  14  20 929
##
## $R_bar
## [1] 1.898226
##
## $Rt
## [1]  0.219634021  1.848627000  0.619102837 ...
```

5. Now implement the non-stationary bandit algorithm in a similar way with parameter α as `nonstationary_algorithm(Q1, bandit, epsilon, alpha)`.

```
set.seed(4711)
Q1 <- rep(0, 5)
nonstationary_algorithm(Q1, stationary_bandit, epsilon = 0.1, alpha = 0.2)
## $Qt
## [1] 1.414101 1.293336 1.237085 0.517214 2.134716
##
## $Nt
## [1] 302  16  19  16 647
##
## $R_bar
## [1] 1.840128
##
## $Rt
## [1]  1.772517635  2.816318235  0.633508626  2.017549413 ...

set.seed(4712)
Q1 <- rep(0, 5)
nonstationary_algorithm(Q1, stationary_bandit, epsilon = 0.1, alpha = 0.2)
## $Qt
## [1] -0.03998096  2.63547855 -0.96654871 -0.56120367  0.09262541
##
## $Nt
## [1] 314 133  38  63 452
##
## $R_bar
## [1] 1.592736
```

```
##
## $Rt
## [1]  0.219634021  1.848627000 ...
```

6. Run each algorithm 500 times/simulations and compute the mean of the mean rewards (for each simulation) for each of the algorithms. Do this for both the stationary and non-stationary bandit. For the non-stationary algorithm, try $\alpha = 0.5$ and $\alpha = 0.9$ and $\epsilon = 0.1$. Similarly, for the ϵ-bandit and use $\epsilon = 0.1$. Summarize all the results in a table. In total you should have a eight different results. *Note!* The algorithm should run 1000 steps for each of the 500 simulations.

7. What are your conclusions from these results? Which algorithm seems to be the best one, and why?

8. Plot the average reward per step for the 500 runs for the worst and best algorithm, such as in Figure 2.2. in Sutton and Barto (2020). Use the output R_t from your implementations.

2 Markov Decision Processes

Now we are going to implement a Markov Decision Process (MDP) where the agent makes a decision, A_t based on the current state S_t and a reward R_t that will, in turn, return a new state S_{t+1} and a reward R_{t+1}. In this part of the assignment, we are going to use the recycling robot example (Example 3.3 in Sutton and Barto, 2020).

1. The MDP in Sutton and Barto (2020) has been implemented as a function `recycling_mdp(alpha, beta, r_wait, r_search)` in the `uuml` R package and can be accessed as follows.

```
library(uuml)
recycling_mdp(0.5, 0.8, 0.1, 1)
```

```
##       s s_next        a p_s_next r_s_a_s_next
## 1  high   high   search      0.5          1.0
## 2  high    low   search      0.5          1.0
## 3   low   high   search      0.2         -3.0
## 4   low    low   search      0.8          1.0
## 5  high   high     wait      1.0          0.1
## 6  high    low     wait      0.0          0.1
## 7   low   high     wait      0.0          0.1
## 8   low    low     wait      1.0          0.1
## 9   low   high recharge      1.0          0.0
## 10  low    low recharge      0.0          0.0
```

2. Now implement a function `always_search_policy(MDP)` that takes an MPD specified above and run the MDP for 1000 steps/actions. The policy should be that the agent always chooses to search, irrespective of the state. The function should return the return divided by the number of time steps and times in each state. The

robot should start in the state `high`. Below is an example of how it should work. Note again, since this is a random algorithm, you might get a slightly different answer.

```
set.seed(4711)
always_search_policy(mdp)
## $Nt
## high  low
##  288  712
##
## $R_bar
## [1] 0.452
```

3. Now implement a function `charge_when_low_policy(MDP)` that takes an MPD specified above and run the MDP for 1000 steps. The policy should be that the agent always chooses to recharge if the energy level is low and always search if the energy level is high. The function should return the same output structure as above.

4. Compare the two policies for MDP with $\alpha = \beta = 0.9$ and $\alpha = \beta = 0.4$ with $r_{\text{wait}} = 0.1$ and $r_{\text{search}} = 1$. Run at least ten times per policy and compute the mean reward. What policy is prefered in the two MDPs?

3 * Some additional bandit algorithms

This task is only necessary to get one point toward a *pass with distinction* (VG) grade. Hence, if you do not want to get a *pass with distinction* (VG) point, you can ignore this part of the assignment.

We are here going to implement two additional bandits and compare them to the previous implementations above.

1. Implement the UCB algorithm as a function called `ucb_algorithm(Q1, bandit, c)`. *Hint!* If $N_t(a) = 0$, then a should be considered as the maximizing action (i.e. should be chosen).

```
set.seed(4711)
Q1 <- rep(0, 5)
ucb_algorithm(Q1, stationary_bandit, c = 2)
## $Qt
## [1]  1.43350715  1.09734192 -0.01287212  0.64762674  2.02954004
##
## $Nt
## [1]  48  26   6  13 907
##
## $R_bar
## [1] 1.946474
##
## $Rt
## [1]  3.43973511  2.57043950  1.89631824 ...
```

```r
set.seed(4712)
Q1 <- rep(0, 5)
ucb_algorithm(Q1, stationary_bandit, c = 2)

## $Qt
## [1] 1.4367906 1.0364514 0.4346884 0.4140891 1.9864393
##
## $Nt
## [1]  57  24  11  10 898
##
## $R_bar
## [1] 1.899517
##
## $Rt
## [1]  1.9818356939  1.1212566109  0.9286269996 ...
```

2. Implement the gradient bandit algorithm as a function called `gradient_bandit_algorithm(bandit, alpha)`. Initialize all $H_t(a) = 1$. The algorithm should print the final preference and the probability of each action instead of Q_t.

```r
set.seed(4711)
gradient_bandit_algorithm(stationary_bandit, alpha = 0.1)
## $Ht
## [1] -0.4771226 -1.0045831 -1.7018038 -1.7427317  4.9262412
##
## $pit
## [1] 0.004 0.003 0.001 0.001 0.990
##
## $Nt
## [1]  27  24  11  12 926
##
## $R_bar
## [1] 1.941705
##
## $Rt
## [1]  1.772517635  2.396318235 -0.286491374 ...

set.seed(4712)
gradient_bandit_algorithm(stationary_bandit, alpha = 0.1)
## $Ht
## [1] -1.217231 -1.201571 -1.938602 -1.241492  5.598897
##
## $pit
## [1] 0.001 0.001 0.001 0.001 0.996
##
## $Nt
## [1]  26  30  18  20 906
##
## $R_bar
## [1] 1.885579
##
## $Rt
## [1]  0.62963402  0.94862700 -0.30089716 ...
```

3. Test to run the gradient bandit with $\alpha = 0$. What happens and why?

4. Run the gradient bandit and the UCB algorithms 500 times/simulations and compute the mean of the mean rewards (for each simulation) for each of the algorithms. *Note!* The algorithm should run 1000 steps for each of the 500 simulations. Use both the stationary and non-stationary bandit. For the UCB, try $c = 0.5$ and $c = 2$ and for the gradient bandit $\alpha = 0.1$ and $\alpha = 1$. Summarize all the results in a table and compare the results with the previous bandit algorithms. Which algorithm performs the best?

References

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction (Second edition)*. MIT press, 2020.