Olga Ibáñez Solé
MA in Bioinformatics and Modeling (ULB)

# MULTI-ARMED BANDITS

We implemented the N-armed bandit problem for N = 4 and studied the effect of three different action selection strategies: random, e-greedy and softmax. We studied the effect of the e parameter in the e-greedy action selection and the t parameter in the softmax strategy. The expected rewards for each four possible actions were initialized to 0 and updated as the agent learned through Q-learning, with a learning rate of alpha = 0.8. The actual rewards were subject to noise and followed a normal distribution of mean *m* and standard deviation *sd*. We averaged the results for each simulation type over 2000 runs of 1000 time steps each.

## Exercise 1

The values of m and sd were the following:

| | m | sd |
|---|---|---|
| Action A | 2.3 | 0.9 |
| Action B | 2.1 | 0.6 |
| Action C | 1.5 | 0.4 |
| Action D | 1.3 | 2.0 |

We run six sets of 2000 simulations, of 1000 time steps each, with the following action selection strategies and parameter values:

- Random
- 0-greedy
- 0.1-greedy
- 0.2-greedy
- Softmax (t=1)
- Softmax (1=0.1)

We plotted the averaged accumulated reward of each simulation type over time (**Figure 1** left). We also plotted the averaged final reward for each simulation type (**Figure 1** right). The color code used in the first figure is respected throughout the whole report. We can observe that the strategy yielding the best accumulated reward was the softmax action selection method with t=1, while the worst was the 0-greedy.

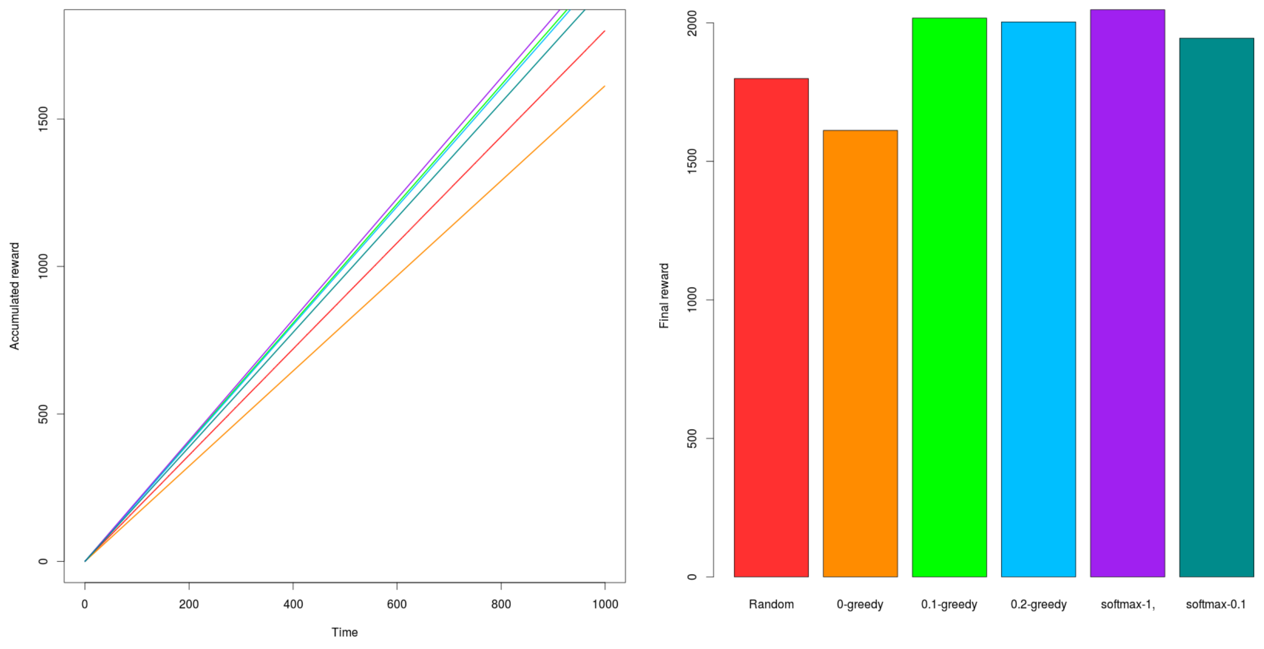

***Figure 1****: accumulated reward over time, averaged over 2000 simulations, for each action selecion strategy (left). Final accumulated reward for each action selection strategy (right).*

We plotted the averaged expected reward of each action over time for each action selection strategy, (**Figure 2**).

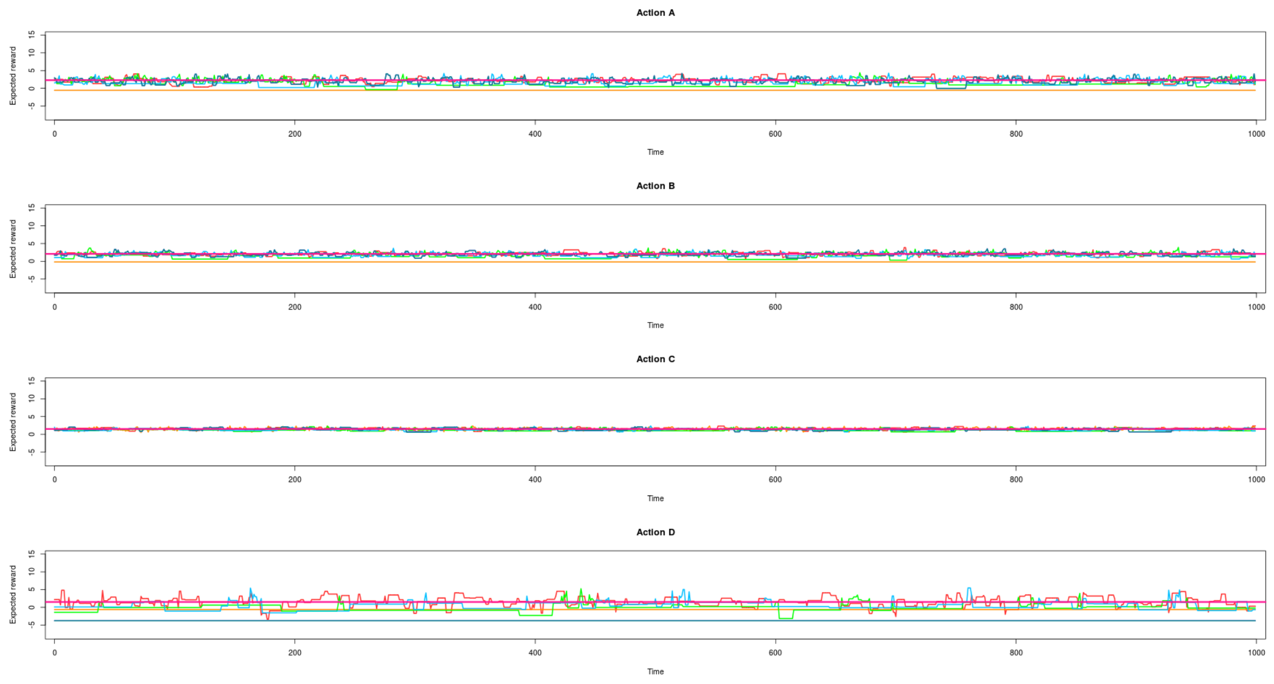

***Figure 2****: expected reward over time for each action and strategy. The pink line shows the actual reward for the corresponding action.*

As we can see in **Figure 2**, as thes agents learn, the expected reward of a given action becomes more similar to the actual reward of that action. As the actual rewards are subject to noise, the agents never stop learning, and the expected rewards oscillate around the mean od the actual reward. The actions for which the reward distributions have a greater standard deviations are the most difficult ones to learn from. In this setting, action D's reward distribution is the one with the greatest standard deviation, and we can clearly see that the agents find more difficulties in guessing the reward of that action. Conversely, action C is the one whose actual reward is less distributed, with a standard deviation of sd = 0.4, and we can see in the plot that the expected rewards are very close to the actual mean reward during the whole simulation. Note: the scale of the plot is kept constant in Exercise 1 and 2 in order to make comparisons easier.

Finally, we plotted the averaged frequencies with which the actions were chosen according to the action selection strategy (**Figure 3**).

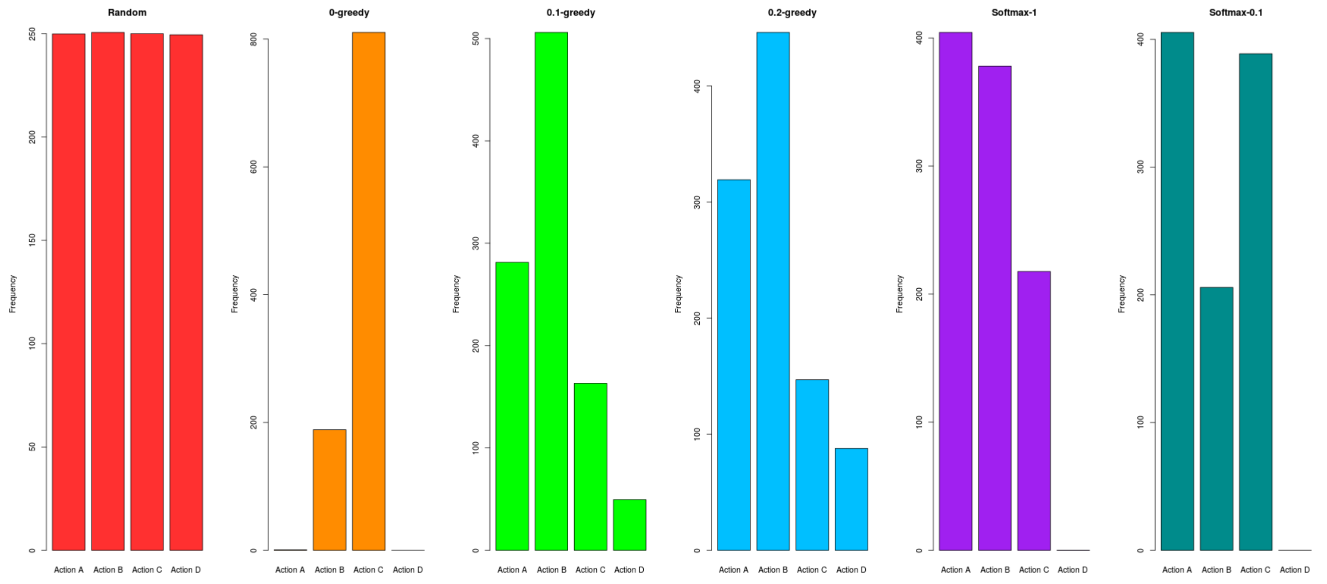

***Figure 3**: averaged frequency with which each action was chosen in each of the simulation types.*

When the actions are chosen randomly, the frequencies are equal for all actions. The agents learn about the rewards of each action but do not exploit this information, instead they keep exploring during the whole simulation.

The 0-greedy is a special case of the e-greedy strategy in wich the agents always choose the greedy action, that is, the one with highest expected reward. At the beginning of the simulation all expected rewards are set to zero, so the agents choose an action randomly. In the next episode, the only action having an expected reward higher than zero is the one that was chosen in the first episode, so the agents keep choosing that action even if it might be suboptimal. For example, in our simulations, action A was rarely chosen even if it was the one with the highest mean reward. This situation reflects the zero-exploration and all-explotation strategy, which has proven to be the one yielding the poores results among the ones studied (**Figure 1**). In the 0.1-greedy strategy, the agents choose randomly in the first episode and then choose the greedy action with probability 0.9 and explore other actions (with equal probability each) with probability 0.1. It proved to be quite good of a strategy, and yielded very similar results as the 0.2-greedy action selection.

The softmax strategy tries to overcome the main distadvantage of e-greedy strategies: the fact that they explore by choosing randomly with equal probabilities for all actions. Softmax strategies rank the actions according to their expected reward and explore with greater probabilties those with highest expected rewards. The parameter t accounts for the "temperature", that is, the relative difference between the probability of exploring actions whose rewards differ. In the limit t=0, the actions are equiprobable and the softmax acts like a greedy action selection strategy. Both softmax with t=1 and softmax with t = 0.1 perfomed very well and chose the best action (A) with the highest frequency. The worst action in terms of mean reward, D, was rarely chosen in either of them. We can see that for t=1, the ranking of frequencies with which each action was chosen match the reward ranking. In other words, the actions with the highest mean reward were chosen more frequently than the actions with poor reward. This makes sense since the temperature was set quite high, which means that the difference between the probability of choosing action A (the best action) and the probability of choosing action D (the worst) was bigger than in the softmax with t=0.1.

## Exercise 2

We doubled the standard deviations of the reward distributions and performed the same simulations as in Exercise 1. The values for m and sd were thus set as follows:

| | m | sd |
|---|---|---|
| Action A | 2.3 | 1.8 |
| Action B | 2.1 | 1.2 |
| Action C | 1.5 | 0.8 |
| Action D | 1.3 | 4.0 |

In **Figure 4** (right) we can observe that all of the strategies yielded poorer final accumulated rewards as compared to Exercise 1. The 0.1-greedy and 0.2-greedy strategies yielded a final reward quite similar to that of random action choice. Softmax with t = 0.1 performed worse than the random action selection. Overall, we can conlude that the effect of increased noise interfered with the learning process of the agents. However, softmax with t=1 was still the best action selection strategy.

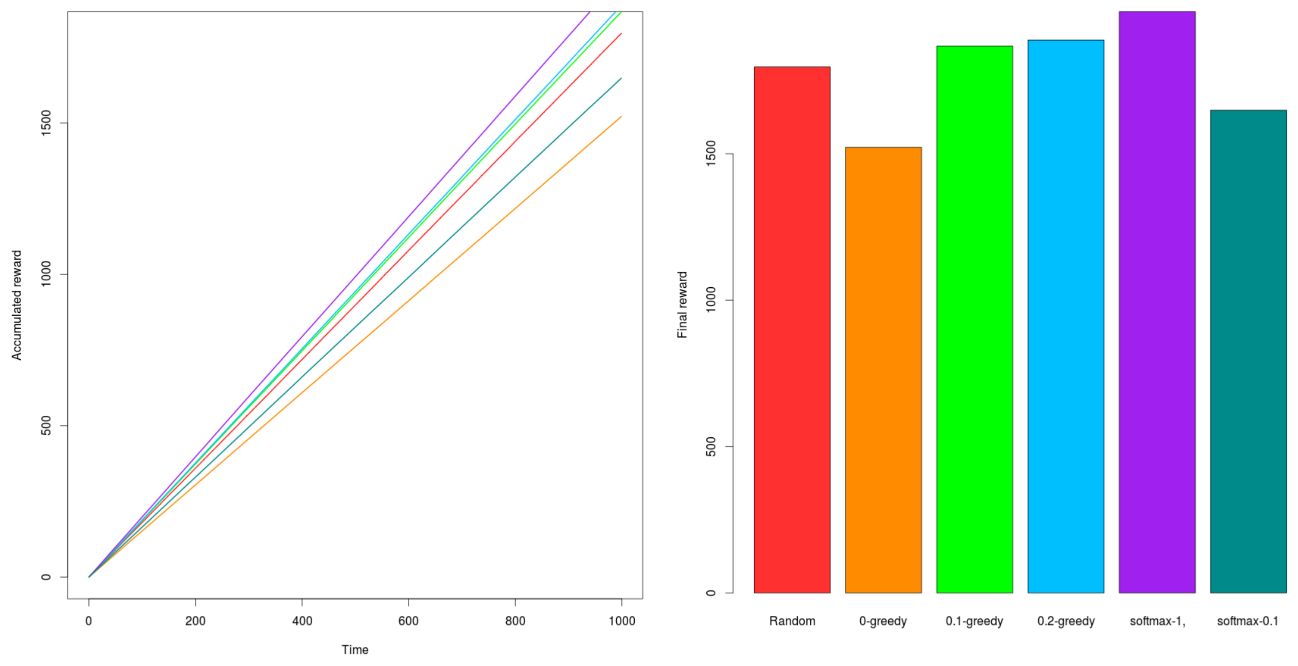

*Figure 4: averaged accumulated reward over time for each strategy (left). Averaged final accumulated reward for each strategy.*

In **Figure 5** we observe that, as in Exercise 1, the actions whose distribution has a greatest standard deviation are the most difficult ones to learn from. We can also see that the irregular oscillations around the mean reward value have a greater amplitude than in **Figure 2**. We can notice that the orange line representing the rewards for actions A, B and D expected by the agents following the strategy 0-greedy did not change over the course of the simulation, as those actions were ignored by the agents. The same holds for softmax learners with t = 0.1, who largely ignored action D.

**Figure 6** shows the frequency with which agents chose each of the strategies. We can notice that 0-greedy learners largely ignored actions A, B and D, as we observed in **Figure 5**. Most importantly, we can clearly see that action A was not the favorite action of any of the learner groups, that is, the best action was missed by all action selection strategies, due to the effect of noise, even by those who were able to identify it in Exercise 1 (Softmax learners with t = 1 and t = 0.1).

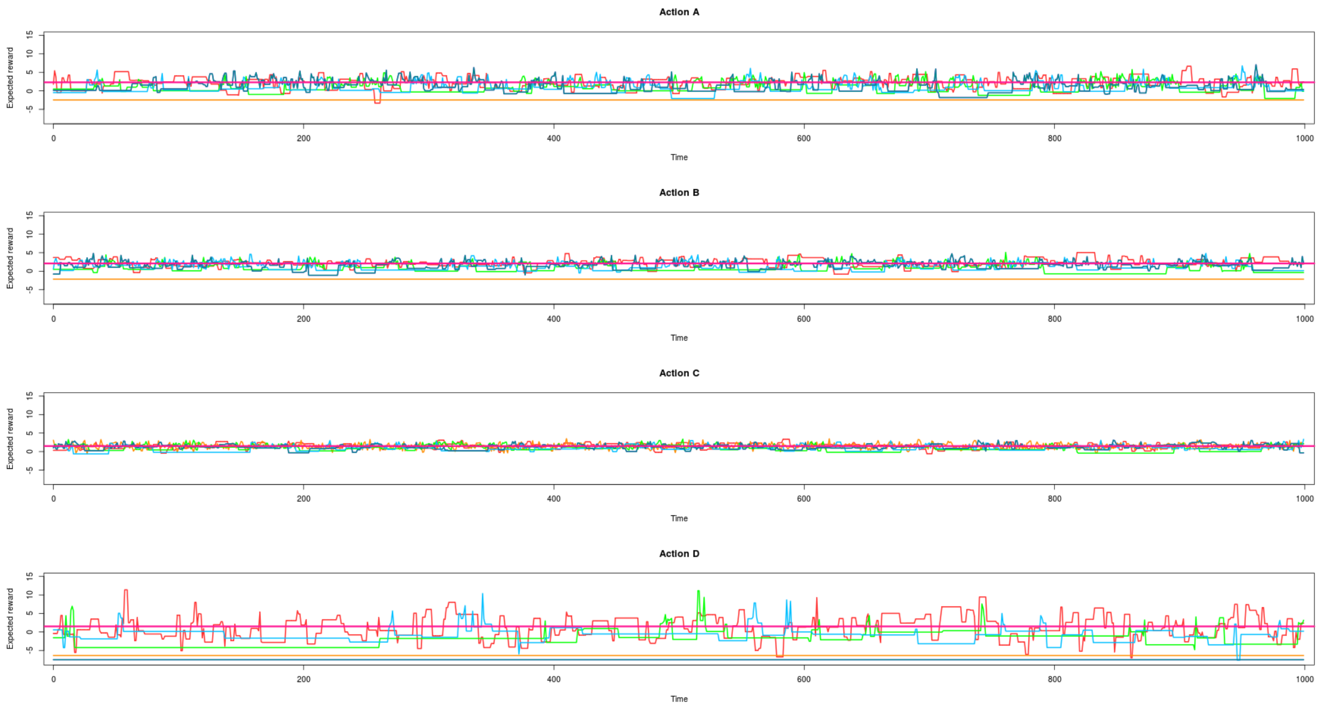

*Figure 5: Averaged expected rewards for each action over time. The pink line represent the actual mean reward for each action.*

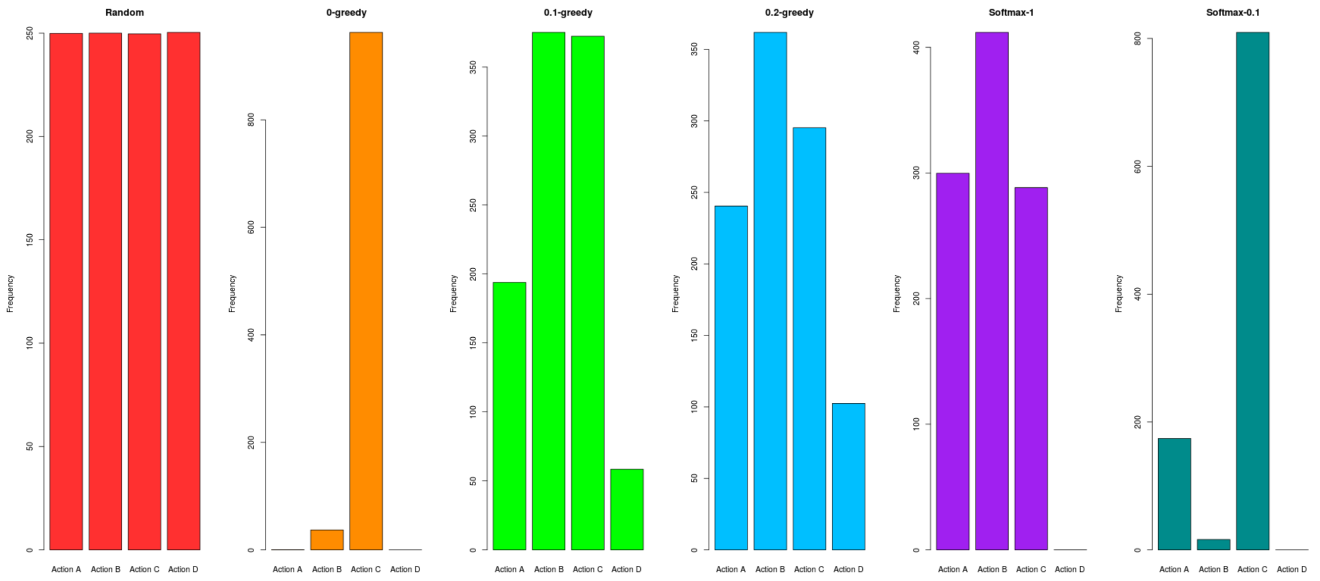

*Figure 6: Frequency with which each action was chosen by the agents following the different action selection strategies.*

## Exercise 3

We performed two additional sets of simulations with a variation of the e-greedy and the softmax action selection strategies. In this variation, the parameters e and t become time-dependent.

e = 1/(time_step**0.5)

t = 4*(1000-time_step)/1000

**Figure 7** shows the values of the time-dependent e and t parameters over time. We can see that t decreases in a linear fashion from t=4 until t=0, and e decreases very rapidly at the beginning and then becomes almost constant.

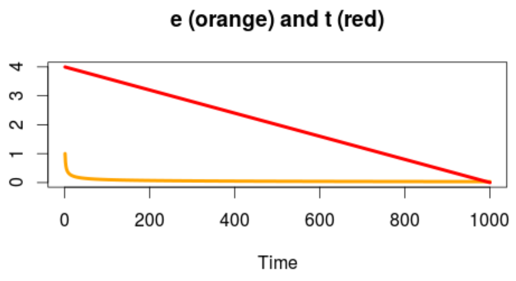

*Figure 7: Time-dependent e and t over time.*

The time-dependent e-value decreases over time, which makes the agents be more explorative at the beginning of the simulation and then become increasingly greedy with time. The advantage of this early exploratory phase is that it prevents the agents from sticking to a suboptimal action and missing a better choice due to a too shallow exploration.

As for the time-dependent t-value, it also decreses over time. Therefore, we would expect the time-dependent softmax to behave as follows: at the beginning of the simulation the temperature would be high, that is to say, the actions would be nearly equiprobable. As time went by, the probabilities of selection of the actions would be increasingly different for those actions whose expected reward differs. Those differences between the selection probabilities would increase over time until t=0, at which the softmax strategy would become equivalent to a greedy action selection. In other words, the decreasing t-value should make the agents more free to explore at the beginning of the simulation and then more conservative (more prone to sticking to those actions having the highest expected reward), as the softmax strategy became increasingly greedy.

In **Figure 8** (left) we can see the averaged accumulated reward for four strategies: 0.2-greedy, softmax with t=1, time-dependent greedy and time-dependent softmax. We can observe that the time-dependent greedy strategy and the 0.2-greedy strategy yield very similar results regarding the averaged accumulated reward. We can also notice that the time-dependent softmax performs worse than the softmax with t=1.

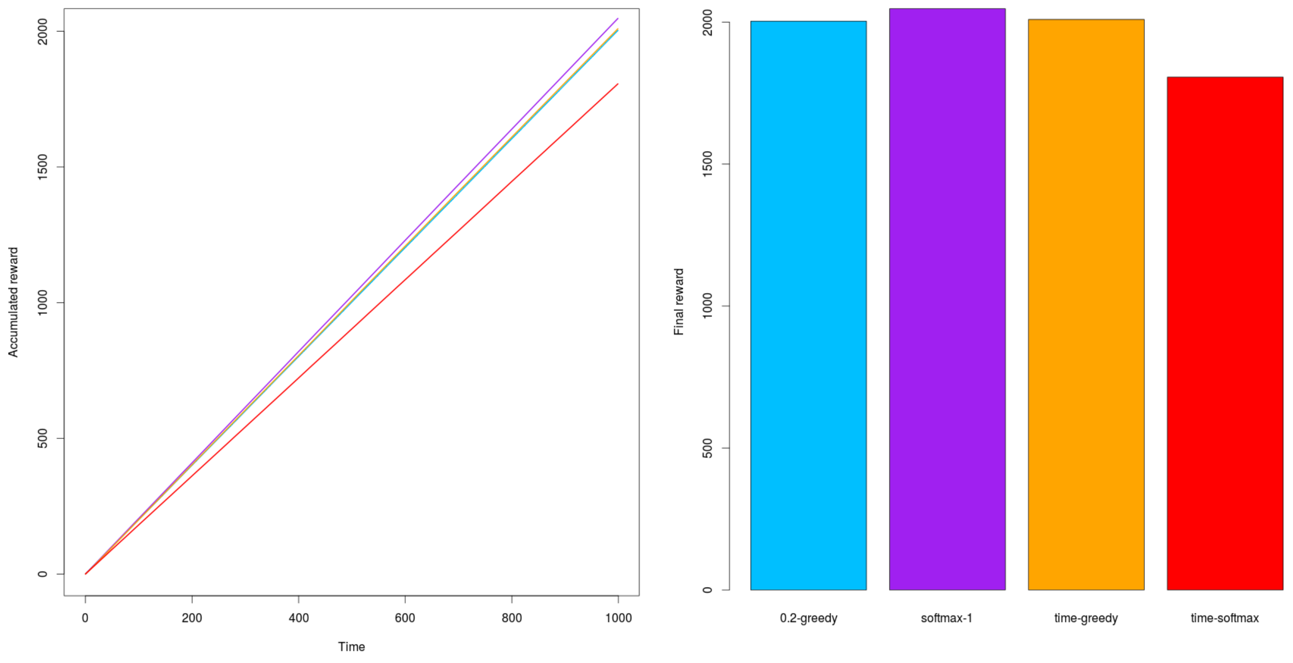

*Figure 8*: *Averaged accumulated reward over time for each action selection strategy (left). Average final accumulated reward for each action selection strategy (right).*

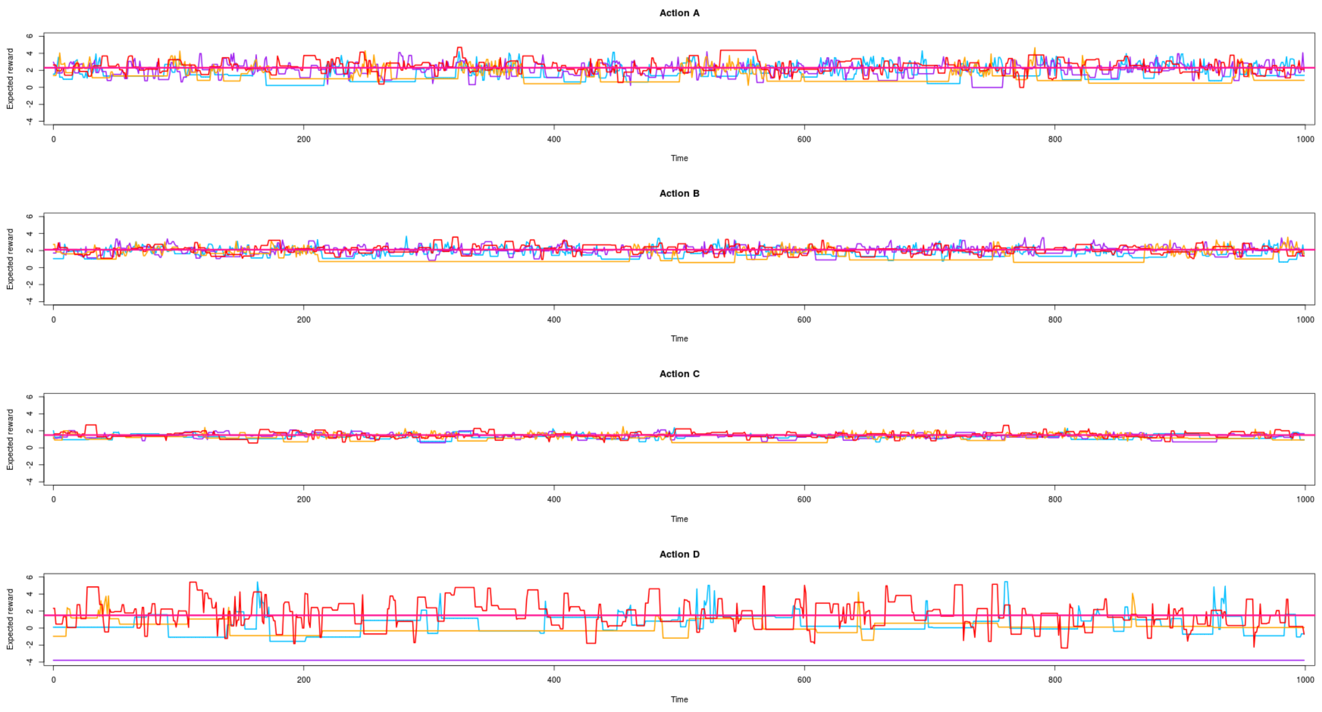

*Figure 9*: *Averaged expected reward for each action over time. The pink line represents the actual mean reward of each action. Note that the scale is different from that of previous equivalent plots.*

In **Figure 9** can observe that the time-dependent softmax (red line) is the action selection strategy that explores action D the most, that is to say, it is the most permisive strategy towards poor expected rewards. Conversely, agents following the softmax with t=1 discard action D from the very beginning and never look back. If we compare the 0.2-greedy with the time-dependent greedy strategy, we can see that the time-dependent greedy is more efficient at discarding poor actions (as D), but also less open to exploration than the 0.2-greedy, and this is why it misses the optimal action A more frequently (**Figure 10**).

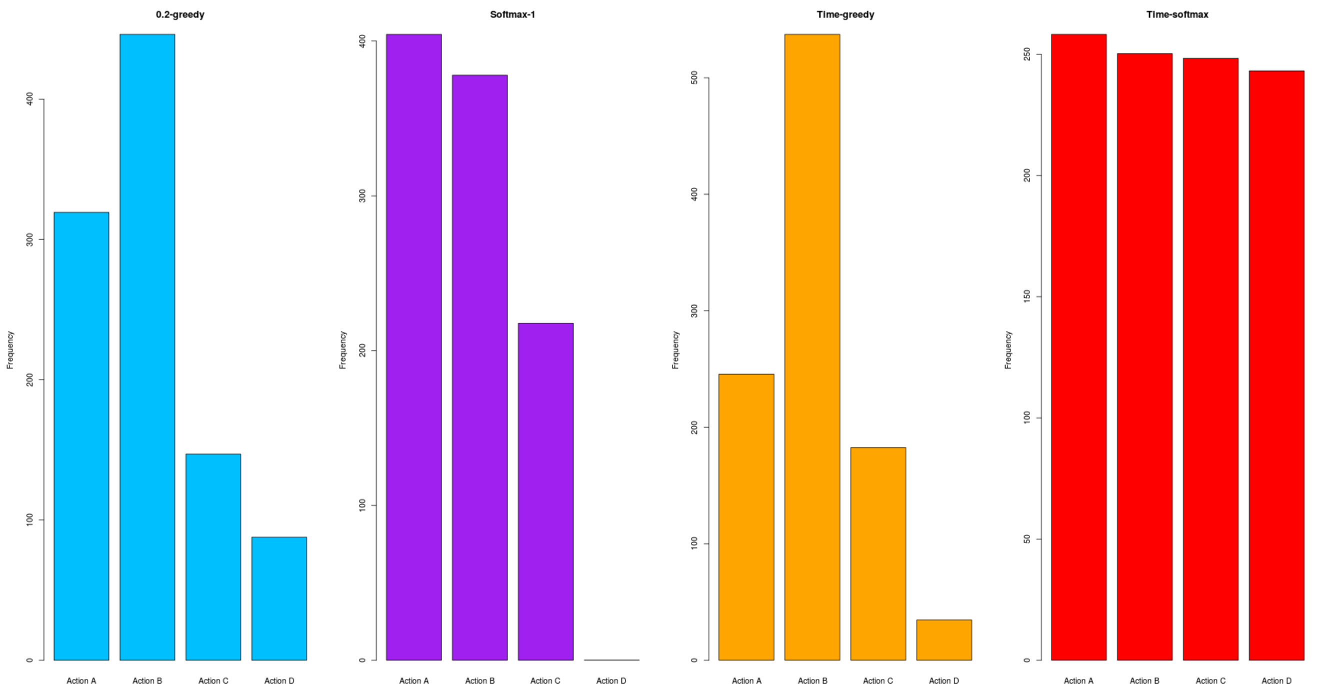

*Figure 10: Frequency with which agents chose each of the actions, for each of the action selection strategies.*

## CODE:

The following Pyhton3 code was used for doing the simulations, and the results were stored in .txt files. The plots that appear throughout the report were built, in R, from those files.

```
import random
import codecs
import numpy as np
from copy import deepcopy

def mean_matrix(matrix_list):
    """
Takes a list of matrices and returns a mean matrix,
that is, a matrix whose elements are the mean of the
elements in the same position in all of the matrices
from the matrix_list.
    """
    mean_matrix = np.mean([i for i in matrix_list], axis=0)
    return mean_matrix

def get_reward(Q, action):
    """
Takes: 1) a list Q containing the mean and standard deviation of the
reward associated to every possible action, and 2) a specific action,
and returns a value for the reward.

    """
    mean = Q[action][0]
    sd = Q[action][1]
    reward = random.normal(mean, sd)

    return reward

def random_action(Q):
    """
Takes the list Q and returns a randomly chosen action.
    """
    action = random.randint(len(Q))

    return action

def greedy_action(Q, e):
    """
Takes the list Q and a parameter e and returns an action
chosen according to the e-greedy selection strategy.
    """
    x = random.randint(100)

    # select random action with probability e:
    if x < 100*e:
        action = random_action(Q)

    # select greedy with probability 1-e:
    else:
        action = q.index(max(q))

    return action

def softmax_action(Q, t):
    """
Takes the list Q and the parameter t and returns
an action chosen according to the softmax selection
strategy.
    """
    upper_terms = [np.e**(q[i]/t) for i in range(len(q))]
    down = sum(upper_terms)
    prob = [up/down for up in upper_terms]
    accum_prob = [sum(prob[:i+1]) for i in range(len(prob))]

    x = random.randint(100)

    if x < accum_prob[0]*100:
        action = 0

    elif x < accum_prob[1]*100:
        action = 1

    elif x < accum_prob[2]*100:
        action = 2

    else:
        action = 3
    return action

def time_dependent_e(time):
    if time == 0:
        time_e = 1.0
    else:
        time_e = 1/(time**0.5)

    return time_e

def time_dependent_t(time):
    if time == 0:
        time_t = 4
    else:
        time_t = 4*(1000*time)/1000

    return time_t

def simulation(time, Q, q, alpha, strategy, e = -1, t = -1):
    """
Takes a number of timesteps time, and a reward list Q.
Returns a matrix containing the expected reward of
each action in every time step.
    """

    taken_actions = len(Q)*[0] # list containing the chosen action
    q_history = []
    accum_rewards = [0]
    reward = 0
    for i in range(time):

        if strategy == "random":
            action = random_action(Q)

        elif strategy == "e-greedy":
            action = greedy_action(Q, e)

        elif strategy == "softmax":
            action = softmax_action(Q, t)

        elif strategy == "time_greedy":
            time_e = time_dependent_e(i)
            action = greedy_action(Q, time_e)

        elif strategy == "time_softmax":
            time_t = time_dependent_t(i)
            action = softmax_action(Q, time_t)

        reward = get_reward(Q, action)
        q[action] = q[action] + alpha*(reward - q[action])

        # update states, q_history and accum_rewards:
        taken_actions[action] += 1
        data = q[:]
        q_history.append(data)
        new_accum_reward = accum_rewards[-1] + reward
        accum_rewards.append(new_accum_reward)

    return taken_actions, q_history, accum_rewards


def simulation_set(n, time, Q, q, alpha, strategy, e= -1, t = -1):
    """
Takes a number of rounds n, a number of timesteps t,
a reward list Q, and an action selection strategy
(random, greedy or softmax).
```

```
    """
    action_avg = len(Q)*[0]
    reward_avg = time*[0]
    for i in range(n):

        q_history_list = []

        if strategy == "random":
            taken_actions, q_history, accum_rewards = simulation(time, Q, q, alpha, "random")


        elif strategy == "e-greedy":
            taken_actions, q_history, accum_rewards = simulation(time, Q, q, alpha, "e-greedy", e)

        elif strategy == "softmax":
            taken_actions, q_history, accum_rewards = simulation(time, Q, q, alpha, "softmax", t = t)

        elif strategy == "time_greedy":
            taken_actions, q_history, accum_rewards = simulation(time, Q, q, alpha, "time_greedy", e =
-1)

        elif strategy == "time_softmax":
            taken_actions, q_history, accum_rewards = simulation(time, Q, q, alpha, "time_softmax", t
= -1)


        action_avg = [action_avg[j] + taken_actions[j] for j in range(len(Q))]
        q_history_list.append(q_history)
        reward_avg = [reward_avg[j] + accum_rewards[j] for j in range(time)]

    # Compute the average taken actions, q_history and accumulated rewards over the n simulations:
    action_avg = [action_avg[i]/n for i in range(len(Q))]
    q_history_avg = mean_matrix(q_history_list)
    reward_avg = [reward_avg[i]/n for i in range(time)]

    filename = str(strategy)+"_(e="+str(e)+",t="+str(t)+")"
    export_results(action_avg, q_history_avg, reward_avg, filename)

    return action_avg, q_history_avg, reward_avg

def export_results(action_avg, q_history_avg, reward_avg, filename):
    """
Takes the mean result matrix and exports it to a .txt file.
This file contains the iteration number, and the expected
rewards for each action for each time-step, averaged over
the n simulation rounds.
    """
    # Actions:
    fn = codecs.open(str(filename)+"_actions", "w", "utf-8")
    fn.write("0\t1\t2\t3\n")
    line = str(action_avg[0])+"\t"+str(action_avg[1])+"\t"+str(action_avg[2])+\
              "\t"+str(action_avg[3])+"\n"
    fn.write(line)
    fn.close()

    # q_history:
    fn = codecs.open(str(filename)+"_q_history", "w", "utf-8")
    fn.write("iteration\t0\t1\t2\t3\n")
    for i in range(len(q_history_avg)):
        line = str(i)+"\t"+str(q_history_avg[i][0])+"\t"+str(q_history_avg[i][1])+"\t"\
              +str(q_history_avg[i][2])+"\t"+str(q_history_avg[i][3])+"\n"
        fn.write(line)
    fn.close()

    # Rewards:
    fn = codecs.open(str(filename)+"_rewards", "w", "utf-8")
    fn.write("iteration\treward\n")
    for i in range(len(reward_avg)):
        line = str(i)+"\t"+str(reward_avg[i])+"\n"
        fn.write(line)
    fn.close()

# Actual rewards:
Q = [(2.3, 0.9), (2.1, 0.6), (1.5, 0.4), (1.3, 2)]
#Q = [(2.3, 1.8), (2.1, 1.2), (1.5, 0.8), (1.3, 4)]

# Expected rewards:
q = [0, 0, 0, 0]

# Learning rate
alpha = 0.8


action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "random")
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "e-greedy", e=0)
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "e-greedy", e=0.1)
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "e-greedy", e=0.2)
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "softmax", t=1)
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "softmax", t=0.1)
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "time_greedy")
action_avg, q_history_avg, reward_avg = simulation_set(2000, 1000, Q, q, alpha, "time_softmax")
```

## Stochastic Reward Game

We implemented two types of Joint Action Learners (JALs) in the stochastic climbing game. JALs learn the value of joint actions, that is, they estimate the reward that they could obtain from combinations of their and the other agents' actions. The matrix below shows the rewars for the joint actions in this game.

| | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|
| $b_1$ | $\mathcal{N}(11, \sigma_0^2)$ | $\mathcal{N}(-30, \sigma^2)$ | $\mathcal{N}(0, \sigma^2)$ |
| $b_2$ | $\mathcal{N}(-30, \sigma^2)$ | $\mathcal{N}(7, \sigma_1^2)$ | $\mathcal{N}(6, \sigma^2)$ |
| $b_3$ | $\mathcal{N}(0, \sigma^2)$ | $\mathcal{N}(0, \sigma^2)$ | $\mathcal{N}(5, \sigma^2)$ |

The stochastic climbing game is known for the difficulties that the agents face in order to achieve coordination (Kapetanakis et al, 2004). This difficulties arise in part because of the stochastic effect, which adds a level of difficulty to any learning process, but also because of the payoff matrix of the game. The penalties that the agents suffer when they do not coordinate are so high that they prevent the agents from learning the optimal coordinated action. Many attempts have been done to overcome this issue, for instance, the introduction of the FMQ action selection strategy, which tries to influence the learners' behavior towards their individual components of the optimal coordinated action. It does so by keeping track of the maximum rewards of every action and how frequently those actions produce their maximum reward. The expected value (EV) is calculated as follows:

EV(action) = Q(action) + c*freq(maxReward(action))*maxReward(action)

The c-value determines the extent to which the FMQ heuristic takes part in the action selection strategy. In the limit case c=0, the EV of the action becomes equal to the regular Q-learning expected reward.

A somewhat easier way to influence learners towards coordination is to implement an optimistic assumption (described by Lauer & Riedmiller, 2000), according to which the Q (action) is only updated if the new value is greater than the current one. We implemented this version of the Joint Action Learners and studied the evolution of rewards over time. We also implemented a regular version of the Join Action Learners in which the actions were selected with a Boltzmann probability that depended on their expected reward (with t = 1). We considered three different sets of values for the standard deviations sd, sd0 and sd1:

a) sd0 = sd1 = sd = 0.2
b) sd0 = 4 and sd1 = sd = 0.1
c) sd1 = 4 and sd0 = sd = 0.1

**Figure 11** shows the reward over time, averaged over 500 simulations, for the two types of learners, with the a), b) and c) values for the standard deviations. Note: we encountered problems when trying to run 5000 simulations in a loop, so we decided to reduce the number of runs. We decided to plot the averaged rewards over time instead of the average accumulated rewards over time because the plots for the accumulated rewards were very similar to the ones shown in the first part of the report. We thus considered that it could be interesting to report the results in a different way, so that the temporal variations over the simulation could also be noticed.

We can observe that the optimistic learner always yields a higher average reward than the regular join action learner. We can also notice that the oscillations are much more irregular and the amplitude is much greater in the B plot, which corresponds to the standard deviations sd0 = 4 and sd1 = sd = 0.1. This suggests that the sd0 has a greater impact on noise (at least on the noise that can make learning difficult) than sd1. If we look at the reward matrix, it is interesting to notice that sd0 is the standard deviation of the reward for the optimal action (with mean = 11).

Although I am uncertain about the interpretation of the results and I cannot draw very firm conclusions, it appears to me that if we considered independent learners (ILs) instead of JALs, the results would not differ much. In fact, Claus and Boutilier (1998) reported that JALs might not take advantage of the extra information that they possess and that they may not outperform ILs.

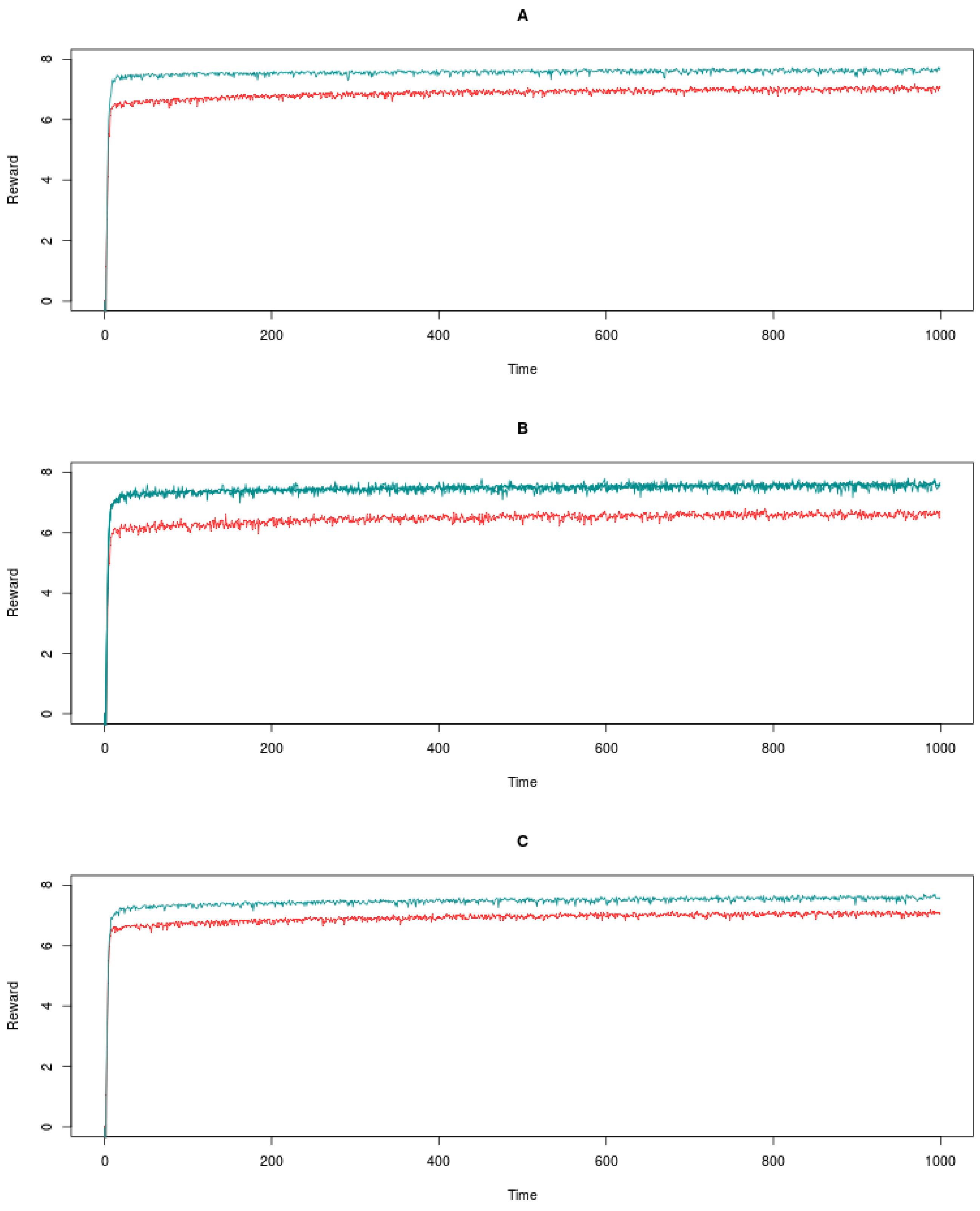

*Figure 11: Averaged reward over time for two typesof learners (JAL in red and optimistic in blue), and for different values of the standard deviations of the rewards (A, B, and C).*

CODE: The following Pyhton3 code was used for doing the simulations, and the results were stored in .txt files. The plots that appear throughout the report were built,  in R, from those files.

```
import numpy as np
import random
import codecs

def initialize_Q_matrix(sd, sd0, sd1):
    """
Creates a 3x3 matrix that contains lists of [mean, variance] of the scores
corresponding to each pair of actions.
    """
    # Initialize matrix:
    Q_matrix = []
    for i in range(3):
        row = []
        for j in range(3):
            row.append([0,0])
        Q_matrix.append(row)

    # Fill in matrix:
    Q_matrix[0][0]  = [11, sd0]
    Q_matrix[0][1] = Q_matrix[1][0] = [-30, sd]
    Q_matrix[0][2] = Q_matrix[2][0] = Q_matrix[2][1] = [0, sd]
    Q_matrix[1][1] = [7, sd1]
    Q_matrix[1][2] = [6, sd]
    Q_matrix[2][2] = [5, sd]

    return np.array(Q_matrix)

def mean_matrix(matrix_list):
    """
Takes a list of matrices and returns a mean matrix,
that is, a matrix whose elements are the mean of the
elements in the same position in all of the matrices
from the matrix_list.
    """
    mean_matrix = np.mean([i for i in matrix_list], axis=0)

    return mean_matrix

def EV(action, q, best, maxR_count, time, c):
    """
Takes an action (action), alist of expected rewards for each action (q),
a list containing the maximum reward obtained so far with each action (best),
a list containing the number of times that the best reward has been
obtained for each action so far (maxR_count), the iteration number (time),
and the parameter c used in the calculation of the expected value through FMQ.

    """
    freq = maxR_count[action]/time
    best_reward = best[action]
    expected_value = q[action] + c*freq*best

    return expected_value

def joint_action(q, tau, belief):
    """
Takes the list Q and the parameter t and returns
an action chosen according to the JAL selection
strategy.
    """
    sumbeliefs = sum(belief)
    if sumbeliefs == 0:
        sumbeliefs = 1
    q2 = [sum([belief[j]*q[i][j] for j in range(3)])/sumbeliefs for i in range(3)]
    upper_terms = [np.e**(q2[i]/tau) for i in range(len(q2))]
    down = sum(upper_terms)
    prob = [up/down for up in upper_terms]
    actions = [0, 1, 2]
    action = np.random.choice(actions, p=prob)
    return action

def get_reward(Q, action1, action2):
    """
Takes: 1) a list Q containing the mean and standard deviation of the
reward associated to every possible action, and 2) two specific actions,
and returns a value for the reward.

    """
    mean = Q[action1][action2][0]
    sd = Q[action1][action2][1]
    reward = random.normalvariate(mean, sd)

    return reward

def simulation(Qa, time, alpha, tau, optimistic = False):
    """
Takes a number of timesteps time, and a reward list Q.
Returns a matrix containing the expected reward of
each action in every time step.
    """
    # Qb is the transposed matrix of Qa:
    Qb = [[Qa[i][j] for i in range(3)] for j in range(3)]

    # Initialize qa and qb matrices:
    qa = np.zeros((3, 3))
    qb = np.zeros((3, 3))

    # Beliefs_
    belief_a = np.zeros(3)
    belief_b = np.zeros(3)

    # Reward lists:
    rewards_a = [0]

    # Accumulate reward lists:
    accum_rewards_a = [0]

    # Rewards:
    reward_a = 0
    reward_b = 0
    for i in range(time):
        action_a = joint_action(qa, tau, belief_a)
        action_b = joint_action(qb, tau, belief_b)
        reward_a = get_reward(Qa, action_a, action_b)
        reward_b = get_reward(Qb, action_b, action_a)

        if not optimistic or alpha*(reward_a - qa[action_a][action_b]) > 0:
            qa[action_a][action_b] += alpha*(reward_a - qa[action_a][action_b])
        if not optimistic or alpha*(reward_b - qb[action_b][action_a]) > 0:
            qb[action_b][action_a] += alpha*(reward_b - qb[action_b][action_a])

        # update beliefs, rewards and accum_rewards:
        belief_a[action_b] += 1
        belief_b[action_a] += 1
        rewards_a.append(reward_a)
        new_accum_reward = accum_rewards_a[-1] + reward_a
        accum_rewards_a.append(new_accum_reward)
    return rewards_a, accum_rewards_a, belief_a, belief_b

def simulation_set(n, Qa, time, alpha, tau):
    """
Takes a number of rounds n, a number of timesteps t,
a reward list Q
    """
    beliefa_avg = np.zeros(3)
    beliefb_avg = np.zeros(3)
    reward_avg = time*[0]

    for i in range(n):
        rewards, accum_rewards, beliefa, beliefb = simulation(Qa, time, alpha, tau)

        actiona_avg = [beliefa_avg[j] + beliefa[j] for j in range(len(Qa))]
        actionb_avg = [beliefb_avg[j] + beliefb[j] for j in range(len(Qa))]

        #reward_avg = [reward_avg[j] + accum_rewards[j] for j in range(time)]

        reward_avg = [reward_avg[j] + rewards[j] for j in range(time)]

    # Compute the average taken actions, q_history and accumulated rewards over the n simulations:
    actiona_avg = [actiona_avg[i]/n for i in range(len(Qa))]
    actionb_avg = [actionb_avg[i]/n for i in range(len(Qa))]
    reward_avg = [reward_avg[i]/n for i in range(time)]

    filename = "stochastic_reward_game_JALC_"
    export_results(actiona_avg, actionb_avg, reward_avg, filename)

    return actiona_avg, actionb_avg, reward_avg, filename

def export_results(actiona_avg, actionb_avg, reward_avg, filename):
    """
Takes the mean result matrix and exports it to a .txt file.
This file contains the iteration number, and the expected
rewards for each action for each time-step, averaged over
the n simulation rounds.
    """
    # Actiona:
    fn = codecs.open(str(filename)+"_actiona", "w", "utf-8")
    fn.write("0\t1\t2\t3\n")
    line = str(actiona_avg[0])+"\t"+str(actiona_avg[1])+"\t"+str(actiona_avg[2])+"\n"
    fn.write(line)
    fn.close()

    # Actionb:
    fn = codecs.open(str(filename)+"_actionb", "w", "utf-8")
    fn.write("0\t1\t2\t3\n")
    line = str(actionb_avg[0])+"\t"+str(actionb_avg[1])+"\t"+str(actionb_avg[2])+"\n"
    fn.write(line)
    fn.close()

    # Rewards:
    fn = codecs.open(str(filename)+"_rewards", "w", "utf-8")
    fn.write("iteration\treward\n")
    for i in range(len(reward_avg)):
        line = str(i)+"\t"+str(reward_avg[i])+"\n"
        fn.write(line)
    fn.close()


#A
#sd = sd0 = sd1 = 0.2

# B:
#sd = sd1 = 0.1
```

```
#sd0 = 4

# C:
sd = sd0 = 0.1
sd1 = 0.4


alpha = 0.8
tau = 1
Qa = initialize_Q_matrix(sd, sd0, sd1)

simulation_set(500, Qa, 1000, alpha, tau)
```