UMEÅ UNIVERSITY **December 16, 2019**
Department of Computing Science
Assignment Report

Artificial Intelligence - Methods and Applications 5DV181

Assignment 3: Multi-Armed Bandits

Name	Jacob Ardnor, Johan Hörnblad
CS Username	c15jar, c15jhd
Version	1.0

1 Comments about the assignment

In the implemented code for this assignment, the multi-armed bandit agents are set to pull one-armed bandits in a setting for 1000 times during 20 session. Each session the expected return of the bandits in the setting are re-randomized, each setting completely independent of the one before. The agents were not reset between each session however. Even if this was intentional, it causes a lot of issues.

Firstly, the total score of the agent stored in a variable called `sums` becomes accumulative since it is not reset between session. This makes counting session wins an ill-suited way of comparing agents because if an agent wins a session, it is more likely to win the one following. Results become very one-sided, especially when the number of sessions is increased, and it becomes very hard to draw conclusions from the result.

Secondly, the internal expected returns of the one-armed bandits does not reset between sessions which causes problems. when a new session is started, the agent still operates on the information from the previous sessions. This information is completely useless because the new setting does not in any way depend on the one preceding it. Some testing we did showed that this can actually seriously hurt agent performance in the new session.

Moreover, each session the agents were given completely different configurations. Some sessions were mathematically unwinnable for our implemented agent just because it had received a setting where the yield of the max-yielding one-armed bandit was too low. It also makes very little sense to compare agent performance in different settings without taking differences in settings into account. Over large amounts of settings this error source becomes less significant, but it is completely unnecessary. Additionally, it makes the fine-tuning of variables during development such as epsilon and decay value very difficult because of the large number of sessions needed to make conclusions.

2 Interpretation

Because of the arguments in the previous section, we decided to slightly modify the code received. In `simulator.py`, the setting randomization part of the `simulate` function was broken out to a new function called `randomize`. These functions were also moved into a new class called `Simulator` which has `expected_rewards_approx` as a field.

In `test_runner.py`, the reference-bandit agent's `sums` field was made sure to reset each iteration, and the setting is first randomized once and then used in the simulation for both agents in each iteration.

3 Approach

The first thing that was implemented to improve the epsilon-greedy algorithm was to replace it with the epsilon-decay algorithm. The epsilon-decay algorithm was constructed by adding a variable x that controlled the amount of exploration, decreasing epsilon by multiplying with x below 1. Just by implementing the epsilon-decay algorithm the implemented agent managed to beat

the reference-bandit agent approximately 9 out of 10 times when the limit to beat the reference bandit was on 5 percent. However When the limit to beat the reference bandit increased the program lost more and more.

During the development, we fine-tuned the values of epsilon and decay value by examining the `frequencies` field of the agent after each session, determining if the agent was able to find the optimal bandit with a low enough number of pulls on the other bandits. Good values for epsilon and the decay value x were found to be $\{\epsilon = 0.1, x = 0.992\}$.

To further improve performance, the memory of the previous round was erased when the agent had pulled one-armed bandits 1000 times. By doing that and reset the score after every iteration the program beat on every round.

4 Results

The final program became an implementation of the epsilon-decay algorithm and With some other improvements that was mentioned earlier like resetting the variables after 1000 iterations and some other small adjustments. The program finally beat the reference-bandit agent every round. Some sessions it was mathematically impossible for our implemented bandit to reach 20% better performance simply because the reference agent did not perform poorly enough.

The number of sessions our implemented agent manages to beat the reference bandit with a 20% margin in a 20-session simulation averages to about 14, where most sessions lost are mathematically unwinnable because of the harsh 20% margin limit.

5 Further improvement suggestions

We found no easy ways to substantially improve the current implemented agent's performance in this environment. By examination of test runs, the agent seems to use a single one-armed bandit the vast majority of the time. Most often, this is the best one-armed bandit. Other times it is the next best bandit, differing from the best bandit by about 0.05 in average yield. The vales of epsilon and its decay value could be fine-tuned further of course.