How to Make Decisions Using Machine Learning Algorithms

By Nosung Myung

Let's assume that we have some sort of idea about the probabilities of the current state, the next state, and the benefits we will receive depending on the final outcome.

A. Maximizing Expected Utility by Choosing the Best Course of Action Given the Probabilities of the Current State and the Next State

1. First, we need to understand the concept of Expected Value of a function of a random variable. In other words, we want to know what kind of result we can expect given an action.

$$E(f(X)) = \sum_{i=1}^{n} p(x_i) f(x_i)$$

```
def get_expectation(x_distribution, function):
    expectation = 0
    for x, p in x_distribution.items():
        expectation = expectation + p * function(x)
    return expectation
```

2. Second, we need to know the Probability of what the next state will be if we perform an action for a certain state. We are interested in how likely a certain result will occur after.

$$P(result(a) = s' | a, e) = \sum_{s} P(result(s, a) = s' | a) P(S_0 = s | e)$$

```
def get_s_prime_probability(s_prime, action, s0_distribution,
transition_table):
    function_s_prime = lambda x: transition_table[x][action][s_prime]
    s_prime_probability = get_expectation(s0_distribution, function_s_prime)
    return s_prime_probability
```

3. Third, we want to figure out the Expected Utility of an action. This will give us how much benefit we will gain from performing an action.

$$EU(a|e) = \sum_{s'} P(result(a) = s' | a, e) U(s')$$

```
def get_eu(s_prime_distribution_given_action, utility_table):
    function_eu = lambda x: s_prime_distribution_given_action[x]
    eu = get_expectation(utility_table, function_eu)
    return eu
```

4. Let's test it with some test data.

```
    def main():
        # s0_distribution: a dictionary representing the probability distribution
of the present state
        s0_distribution = {0:0.125, 1:0.25, 2:0.0625, 3:0.0625, 4:0.25, 5:0.25}

        # utility_table: a dictionary representing the utility of each state
        utility_table = {0:2000, 1:-500, 2:100, 3:1000, 4:0, 5:-5000}

        # transition_table: a dictionary showing the possible results of actions
        # {s0:{a:{s':p}}}: p is the probability of the event (Start from s0, take
    action a, end up at s')
        transition_table = {0:{0:{0:1,1:0,2:0,3:0,4:0,5:0},
                               1:{0:0.05,1:0.9,2:0.05,3:0,4:0,5:0},
                               2:{0:0,1:0.05,2:0.9,3:0.05,4:0,5:0},
                               3:{0:0,1:0,2:0.05,3:0.9,4:0.05,5:0},
                               4:{0:0,1:0,2:0,3:0.05,4:0.9,5:0.05},
                               5:{0:0.05,1:0,2:0,3:0,4:0.05,5:0.9}},
                           1:{0:{0:0,1:1,2:0,3:0,4:0,5:0},
                               1:{0:0,1:0.05,2:0.9,3:0.05,4:0,5:0},
                               2:{0:0,1:0,2:0.05,3:0.9,4:0.05,5:0},
                               3:{0:0,1:0,2:0,3:0.05,4:0.9,5:0.05},
                               4:{0:0.05,1:0,2:0,3:0,4:0.05,5:0.9},
                               5:{0:0.9,1:0.05,2:0,3:0,4:0,5:0.05}},
                           2:{0:{0:0,1:0,2:1,3:0,4:0,5:0},
                               1:{0:0,1:0,2:0.05,3:0.9,4:0.05,5:0},
                               2:{0:0,1:0,2:0,3:0.05,4:0.9,5:0.05},
                               3:{0:0.05,1:0,2:0,3:0,4:0.05,5:0.9},
                               4:{0:0.9,1:0.05,2:0,3:0,4:0,5:0.05},
                               5:{0:0.05,1:0.9,2:0.05,3:0,4:0,5:0}},
                           3:{0:{0:0,1:0,2:0,3:1,4:0,5:0},
                               1:{0:0,1:0,2:0,3:0.05,4:0.9,5:0.05},
                               2:{0:0.05,1:0,2:0,3:0,4:0.05,5:0.9},
                               3:{0:0.9,1:0.05,2:0,3:0,4:0,5:0.05},
                               4:{0:0.05,1:0.9,2:0.05,3:0,4:0,5:0},
                               5:{0:0,1:0.05,2:0.9,3:0.05,4:0,5:0}},
                           4:{0:{0:0,1:0,2:0,3:0,4:1,5:0},
                               1:{0:0.05,1:0,2:0,3:0,4:0.05,5:0.9},
                               2:{0:0.9,1:0.05,2:0,3:0,4:0,5:0.05},
                               3:{0:0.05,1:0.9,2:0.05,3:0,4:0,5:0},
                               4:{0:0,1:0.05,2:0.9,3:0.05,4:0,5:0},
                               5:{0:0,1:0,2:0.05,3:0.9,4:0.05,5:0}},
                           5:{0:{0:0,1:0,2:0,3:0,4:0,5:1},
                               1:{0:0.9,1:0.05,2:0,3:0,4:0,5:0.05},
                               2:{0:0.05,1:0.9,2:0.05,3:0,4:0,5:0},
                               3:{0:0,1:0.05,2:0.9,3:0.05,4:0,5:0},
                               4:{0:0,1:0,2:0.05,3:0.9,4:0.05,5:0},
                               5:{0:0,1:0,2:0,3:0.05,4:0.9,5:0.05}}}

        action_space = [0,1,2,3,4,5]
        state_space = [0,1,2,3,4,5]
```

```
        s_prime_distribution = {a: {s: get_s_prime_probability(s, a,
s0_distribution, transition_table) for s in state_space} for a in action_space}
        eu = {a: get_eu(s_prime_distribution[a], utility_table) for a in
action_space}

        print(eu)
    # {0: -1056.25, 1: -689.0624999999999, 2: 248.12499999999994, 3:
-174.06250000000003, 4: -796.25, 5: -6.875}
    # We can see that taking action 2 gives us the best expected utility given the
probabilities of two things: the current state and the next state given an action.

    if __name__ == '__main__':
        main()
```

5. Conclusion: Taking Action 2 yields the most benefit (Expected Utility) with the given information we have.

```
{0: -1056.25, 1: -689.0625, 2: 248.125, 3: -174.0625, 4: -796.25, 5: -6.875}
```

It would be nice to have all of the information prior to making any decisions. However, in real-life situations, that is not always the case. Now, we are going to assume we have no prior information about anything. Thus, we will "explore" first and then "exploit" options that have the most benefit.

B. Maximizing Reward Without Any Prior Information by Exploring and Exploiting With the Multi-Armed Bandit Reinforcement Learning Technique and the Epsilon-Greedy and Upper Confidence Bound Algorithms

1. Epsilon-Greedy. We will assume that we have no prior information. What we will do is try each action once to gather some data. Then, we will choose whichever action yields the best result with a probability of 1 - epsilon or at random with a probability of epsilon. After each action, we will update the reward for each action accordingly. We will try multiple epsilons to see which one yields the best result.

$$A \leftarrow \begin{cases} arg\ max_a Q(a), with\ probability\ 1-\epsilon \\ random\ action, with\ probability\ \epsilon \end{cases}$$

```
import numpy as np
import matplotlib.pyplot as plt

# Q: dictionary with keys as actions and values as average reward
# e: scalar that is used to determine the degree of randomness

def e_greedy(Q, e):
    p = np.random.random()
    # checking if we will select at random
```

```
    if p <= e:
        random_select = np.random.randint(len(Q))
        action = list(Q)[random_select]
    else:
        action_max = max(Q.values())
        action_best = [action for action, reward in Q.items() if
reward == action_max]
        action = np.random.choice(action_best)
    return action
```

2. Upper Confidence Bound. Choosing the current best option and randomizing from time to time is a great idea. However, once an option gets selected as the best option, it may become highly biased towards it. Thus, we want to also take into account how many times an option has been selected so that we can offset the imbalance.

$$A(t) = arg\,max_a \left[Q(t) + c\sqrt{\frac{ln(t)}{N_t(a)}} \right]$$

```
# N: dictionary with keys as actions and values as the frequency
# c: scalar that can be used to put varying weights on the frequency

def upper_confidence_bound(Q, N, c):
    t = sum(list(N.values())) + 1
    if 0 in list(N.values()):
        action_best = [k for k, v in N.items() if v == 0]
        action = np.random.choice(action_best)
    else:
        bound = [q + c * np.sqrt(np.log(t) / n) for q, n in
zip(list(Q.values()), list(N.values()))]
        bound_max = max(bound)
        action_best = [q for q, b in zip(Q.keys(), bound) if b ==
bound_max]
        action = np.random.choice(action_best)
    return action
```

Now that we have a function for the algorithm, we need an auxiliary function to update the Q and N dictionaries.

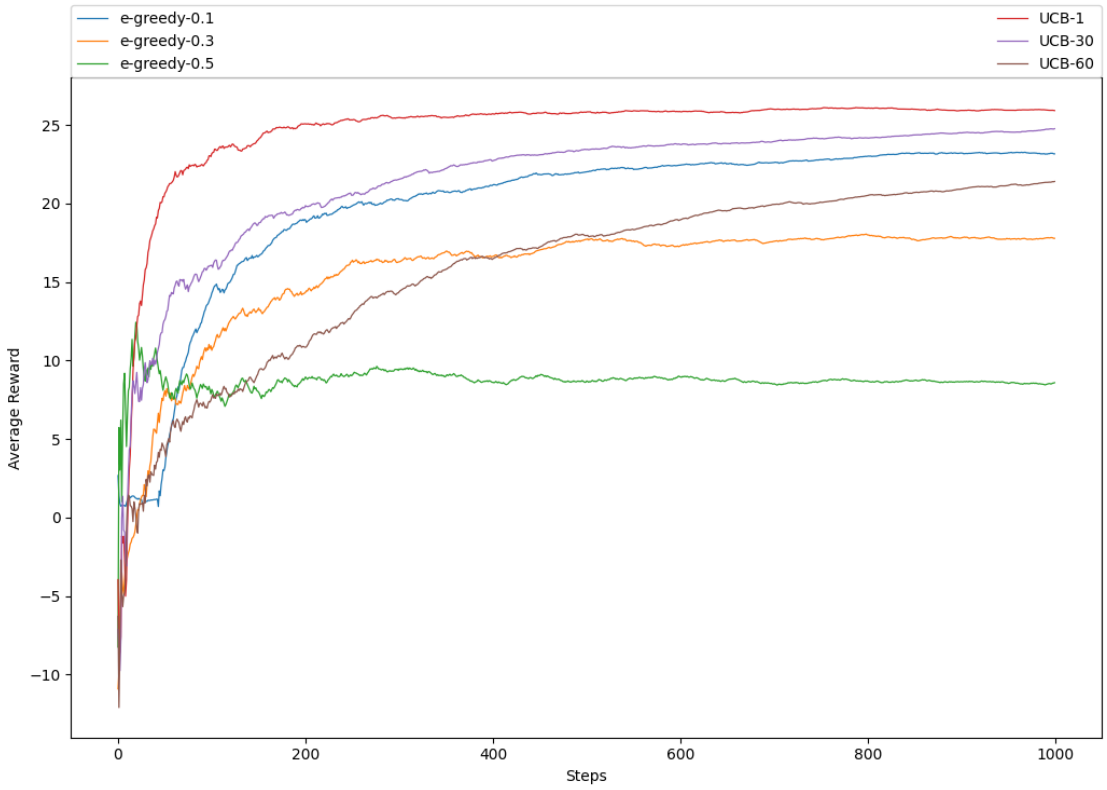
```
def update_QN(action, reward, Q, N):
    N_new = N.copy()
    N_new[action] = N_new[action] + 1
    Q_new = Q.copy()
    Q_new[action] = Q_new[action] + 1/N_new[action] * (reward -
Q_new[action])
    return Q_new, N_new
```

3. Let's now test the algorithms and see which one performs the best.

```
def decide_multiple_steps(Q, N, policy, bandit, max_steps):
    action_reward = []
    for i in range(max_steps):
        action = policy(Q, N)
        reward = bandit(action)
        action_reward.append((action, reward))
        Q, N = update_QN(action, reward, Q, N)
    return {'Q': Q, 'N': N, 'action_reward': action_reward}

def plot_mean_reward(action_reward,label):
    max_steps = len(action_reward)
    reward = [reward for (action,reward) in action_reward]
    mean_reward = [sum(reward[:(i+1)])/(i+1) for i in
range(max_steps)]
    plt.plot(range(max_steps), mean_reward, linewidth = 0.9, label =
label)
    plt.xlabel('Steps')
    plt.ylabel('Average Reward')

def get_samplar():
    mu = np.random.normal(0, 10)
    sd = abs(np.random.normal(5, 2))
    get_sample = lambda: np.random.normal(mu, sd)
    return get_sample

def main():
    np.random.seed(2023)
    K = 10
    max_steps = 1000
    Q = {k: 0 for k in range(K)}
    N = {k: 0 for k in range(K)}
    test_bed = {k: get_samplar() for k in range(K)}
    bandit = lambda action: test_bed[action]()

    policies = {}
    policies["e-greedy-0.1"] = lambda Q, N: e_greedy(Q, 0.1)
    policies["e-greedy-0.3"] = lambda Q, N: e_greedy(Q, 0.3)
    policies["e-greedy-0.5"] = lambda Q, N: e_greedy(Q, 0.6)
    policies["UCB-1"] = lambda Q, N: upper_confidence_bound(Q, N, 1)
    policies["UCB-30"] = lambda Q, N: upper_confidence_bound(Q, N,
30)
    policies["UCB-60"] = lambda Q, N: upper_confidence_bound(Q, N,
60)
```

```
    all_results = {name: decide_multiple_steps(Q, N, policy, bandit,
max_steps) for (name, policy) in policies.items()}

    for name, result in all_results.items():
         plot_mean_reward(all_results[name]['action_reward'], label
= name)
    plt.legend(bbox_to_anchor = (0, 1, 1, 0), loc = 'lower left',
ncol = 2, mode = "expand", borderaxespad = 0)
    plt.show()

if __name__=='__main__':
    main()
```

4. For the Epsilon-Greedy algorithm, having a large epsilon doesn't seem to help too much in the long run. It does seem to outperform at the beginning when there is less information. For the Upper Confidence Bound algorithm, although having a smaller weight scalar c seems to perform the best, they all seem to be converging as the trial numbers grow. Overall, the Upper Confidence Bound algorithm outperforms the Epsilon-Greedy algorithm for obvious reasons.



Initially, we assumed that we knew the probabilities of the future states and rewards and made decisions based off of that. Then, we assumed that we had no information prior to making any decisions. However, we live in an age where data is abundant and easily accessible, though it may not always be accurate. We will take advantage of this fact and utilize it to make decisions by predicting the future probabilities and rewards as we go to make the most efficient decisions. At the end of the day, aren't making decisions that are the most rewarding and predictable using all the resources we have the best type of decision we can make? Let's go through how we can do this.

C. Maximizing Reward Using the Bellman Equation and Dynamic Programming With Value Iteration and Policy Iteration Algorithms to Find Optimal Solutions for Different Settings

1. First, we will take a look at the Bellman Equation for a policy. The equation looks daunting at first, but what it tells you is that the value of a state from a policy is the sum of immediate reward of transitioning to the state and the discounted value of the next state.

$$v_{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_{\pi}(s')\Big]$$

From the Bellman equation, we will look at the second portion of the equation which is often called the Q-value. As we mentioned, the Q-value represents the value of the next state. It tells us how good of a transition to the next state will be, which ultimately tells us what kind of decision we should make.

$$Q(s, a) = \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \, max_{a'} Q(s', a')\Big]$$

We will write a function to update the Q along with some helper functions similar to what we saw earlier.

```
def expect(x_distribution, function):
    expectation = sum([function(x) * px for x, px in
x_distribution.items()])
    return expectation

def get_s_prime_distribution_full(s, action, transition_table,
reward_table):
    reward = lambda s_prime: reward_table[s][action][s_prime]
    p = lambda s_prime: transition_table[s][action][s_prime]
    s_prime_distribution = {(s_prime, reward(s_prime)): p(s_prime)
for s_prime in transition_table[s][action].keys()}
```

```
    return s_prime_distribution

def update_Q_full(s, a, Q, get_s_prime_distribution, gamma):
    s_prime_distribution = get_s_prime_distribution(s, a)
    Q_s_a = sum([p_sp * (sp[1] + gamma * max(Q[sp[0]].values())) for
sp, p_sp in s_prime_dist.items()])
    return Q_s_a
```

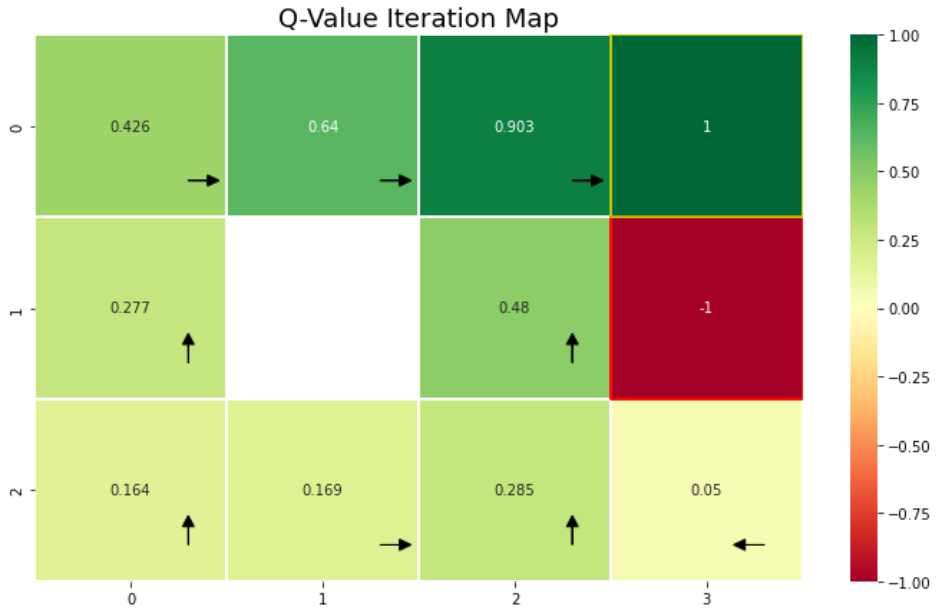
2. Second, we want to perform the Value Iteration algorithm with the Q-value using dynamic programming.

```
def q_value_iteration(Q, update_Q, state_space, action_space,
convergence_tolerance):
    while True:
        par = 0
        Q_new = Q
        for s in state_space:
            for a in action_space:
                if abs(update_Q(s, a, Q) - Q[s][a]) >
convergence_tolerance:
                    Q_new[s][a] = update_Q(s, a, Q_new)
                    par = par + 1
        if par == 0:
            break
    return Q_new
```

3. Lastly, we want to derive policies by using the Q-value table to find the optimal policy for maximum reward.

```
def get_policy_full(Q, rounding_tolerance):
    max_val = max(Q.values())
    policy_val = [k for k, v in Q.items() if abs(v - max_val) <
rounding_tolerance]
    policy = {p: (1 / len(policy_val)) for p in policy_val}
    return policy
```

4. Let's see what this looks like with an example. The example below is a model that represents what the expected reward would be depending on where an agent started its journey. The agent can move up, right, or left with a certain probability distribution and each move will cost a certain amount. They can end up in either of the terminal states, the +1 state or the -1 state. After numerous iterations, the map below has been created. The numbers indicate what the expected reward would be depending on their initial positions and the arrows show the optimal path to take to achieve maximum reward.




```
import numpy as np
import draw_heat_map as hm
import reward_table as rt
import transition_table as tt


def main():

    min_x, max_x, min_y, max_y = (0, 3, 0, 2)

    action_space = [(0,1), (0,-1), (1,0), (-1,0)]
    state_space = [(i, j) for i in range(max_x + 1) for j in
range(max_y + 1) if (i, j) != (1, 1)]
    Q = {s: {a: 0 for a in action_space} for s in state_space}

    normal_cost = -0.04
    trap_dict = {(3,1): -1}
    bonus_dict = {(3,0): 1}
    block_list = [(1, 1)]

    p = 0.8
    transition_probability = {'forward': p, 'left': (1-p) / 2,
'right': (1-p) / 2, 'back': 0}
    transition_probability = {move: p for move, p in
transition_probability.items() if transition_probability[move] != 0}

    transition_table = tt.create_transition_table(min_x, min_y,
max_x, max_y, trap_dict, bonus_dict, block_list, action_space,
transition_probability)
```

```
    reward_table = rt.create_reward_table(transition_table,
normal_cost, trap_dict, bonus_dict)

    s_prime_distribution = lambda s, action:
get_s_prime_distribution_full(s, action, transition_table,
reward_table)
    gamma = 0.8
    update_Q = lambda s, a, Q: update_Q_full(s, a, Q,
s_prime_distribution, gamma)

    convergence_tolerance = 1e-7
    Q_new = Q_value_iteration(Q, update_Q, state_space,
action_space, convergence_tolerance)

    rounding_tolerance = 1e-7
    get_policy = lambda Q: get_policy_full(Q, rounding_tolerance)
    policy = {s: get_policy(Q_new[s]) for s in state_space}

    V = {s: max(Q_new[s].values()) for s in state_space}

    V_drawing = V.copy()
    V_drawing[(1, 1)] = 0
    V_drawing = {k: v for k, v in sorted(V_drawing.items(), key =
lambda item: item[0])}
    policy_drawing = policy.copy()
    policy_drawing[(1, 1)] = {(1, 0): 1.0}
    policy_drawing = {k: v for k, v in
sorted(policy_drawing.items(), key = lambda item: item[0])}

    hm.draw_final_map(V_drawing, policy_drawing, trap_dict,
bonus_dict, block_list, normal_cost)

if __name__=='__main__':
    main()
```