Report Assignment 1

Thi Tuyet Nhi Nguyen

Faculty of Applied Science, Memorial University of Newfoundland

The code is available at: https://github.com/snoween/Reinforcement_Learning/tree/main/Assignment1

I. Part 1

To do this assignment, first need to import necessary libraries as below:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

1. Setting up a simple bandit problem with stationary reward distributions

To reuse the code for the whole assignment, we will generate a class *'StationaryBandit'* to set up a simple bandit problem with stationary reward distributions. The function of this class is to generate a set of ten iid means μ1,...,μ10 from a N(0,1) distribution and suppose that the arms 1 through 10 have N(μi,1) reward distributions where i = 1,...,10.

```
# Bandit environment
class StationaryBandit:
    def __init__(self, k=10):
        self.k = k
        self.true_means = np.random.normal(0, 1, size=k)
        self.optimal_action = np.argmax(self.true_means)

    def pull(self, action):
        return np.random.normal(self.true_means[action], 1)
```

In the next sessions, we will write programs for four algorithms, including: greedy with non-optimistic initial values, epsilon-greedy with different choices of epsilon, optimistic starting values with a greedy approach, and gradient bandit algorithm. To do that, we need to generate base class for algorithms:

```
# Base class for algorithms
class BanditAlgorithm:
    def __init__(self, k):
        self.k = k
        self.counts = np.zeros(k)
        self.q_values = np.zeros(k)

    def update(self, action, reward):
        self.counts[action] += 1
        alpha = 1 / self.counts[action]
        self.q_values[action] += alpha * (reward - self.q_values[action])
```

2. Greedy Algorithm

The code for greedy with non-optimistic initial values, i.e. 0

```
# 1. Greedy (non-optimistic)
class Greedy(BanditAlgorithm):
    def select_action(self):
        max_value = np.max(self.q_values)
        candidates = np.where(self.q_values == max_value)[0]
        return np.random.choice(candidates)
```

3. Epsilon-Greedy Algorithm

The code for epsilon-greedy with different choices of epsilon.

```
# 2. Epsilon-greedy
class EpsilonGreedy(BanditAlgorithm):
    def __init__(self, k, epsilon):
        super().__init__(k)
        self.epsilon = epsilon

    def select_action(self):
        if np.random.rand() < self.epsilon:
            return np.random.randint(self.k)
        else:
            max_value = np.max(self.q_values)
            candidates = np.where(self.q_values == max_value)[0]
            return np.random.choice(candidates)
```

To select the value of ϵ. We use pilot runs: small-scale experiments to try a grid of several settings at low computation cost, tracking the evolution of the rewards curve for each setting to pick a setting that gives good results. The code for pilot-run is as below:

```
# Pilot run for epsilon-greedy
def pilot_test_epsilons(epsilons, n_runs=100, n_steps=500):
    plt.figure(figsize=(12, 6))
    for eps in epsilons:
        rewards, _ = run_experiment(EpsilonGreedy, {"k": 10, "epsilon": eps},
                                    n_steps=n_steps, n_runs=n_runs)
        plt.plot(rewards, label=f"epsilon={eps}")
    plt.title("Pilot run: Average rewards for different epsilon values")
    plt.xlabel("Time Step")
    plt.ylabel("Average Reward")
    plt.legend()
    plt.grid()
    plt.show()
```

The result will be:

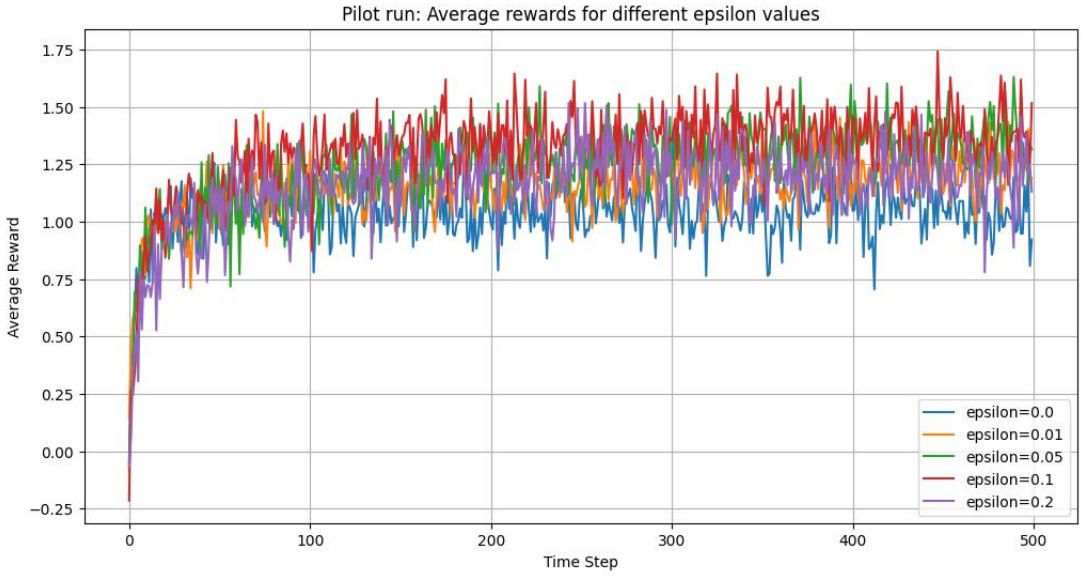

Figure 1: The pilot run to choose the best epsilon value

The chart shows that the best epsilon value should be 0.1. So, we will take this epsilon value for the remaining code.

4. Optimistic Greedy Algorithm

The code for optimistic greedy algorithm.

```
# 3. Optimistic greedy
class OptimisticGreedy(BanditAlgorithm):
    def __init__(self, k, optimistic_value):
        super().__init__(k)
        self.q_values.fill(optimistic_value)

    def select_action(self):
        max_value = np.max(self.q_values)
        candidates = np.where(self.q_values == max_value)[0]
        return np.random.choice(candidates)
```

5. Gradient Bandit Algorithm

The code for gradient bandit algorithm.

```
# 4. Gradient bandit
class GradientBandit:
    def __init__(self, k, alpha):
        self.k = k
        self.alpha = alpha
        self.preferences = np.zeros(k)
        self.avg_reward = 0
        self.step = 0

    def select_action(self):
        exp_prefs = np.exp(self.preferences)
        self.probs = exp_prefs / np.sum(exp_prefs)
        return np.random.choice(self.k, p=self.probs)

    def update(self, action, reward):
        self.step += 1
        self.avg_reward += (reward - self.avg_reward) / self.step
        for a in range(self.k):
            if a == action:
                self.preferences[a] += self.alpha * (reward - self.avg_reward) *
                (1 - self.probs[a])
            else:
                self.preferences[a] -= self.alpha * (reward - self.avg_reward) *
                self.probs[a]
```

6. Implement Algorithms

In this part, we need to run each algorithm for 2000 steps. Repeat for a total of 1000 simulations and report (1) the average reward acquired by the algorithm at each time step, averaged over 1000 simulations and (2) the percentage of time the optimal action is taken by the algorithm at each time step. The code for implementation:

```python
# Experiment runner
def run_experiment(algo_class, algo_args={}, n_steps=2000, n_runs=1000):
    rewards = np.zeros(n_steps)
    optimal_actions = np.zeros(n_steps)

    for run in range(n_runs):
        env = StationaryBandit()
        algo = algo_class(**algo_args)
        for t in range(n_steps):
            action = algo.select_action()
            reward = env.pull(action)
            algo.update(action, reward)
            rewards[t] += reward
            if action == env.optimal_action:
                optimal_actions[t] += 1

    rewards /= n_runs       #Average each 1000 simulations
    optimal_actions = 100 * optimal_actions / n_runs
    return rewards, optimal_actions
```

To run the code for Part 1, we call all the above classes:

```python
# Main
if __name__ == "__main__":
    k = 10
    mu_max = 2  # Reasonable upper bound for optimistic initialization
    optimistic_value = norm(loc=mu_max, scale=1).ppf(0.995)

    # Run pilot test for epsilon-greedy
    epsilons = [0.0, 0.01, 0.05, 0.1, 0.2]
    pilot_test_epsilons(epsilons, n_runs=100, n_steps=500)

    # Final experiments
    methods = [
                ("Greedy", Greedy, {"k": k}),
                ("Epsilon-Greedy (0.1)", EpsilonGreedy, {"k": k, "epsilon": 0.1}),
                ("Optimistic Greedy", OptimisticGreedy, {"k": k,
                "optimistic_value": optimistic_value}),
                ("Gradient Bandit (0.1)", GradientBandit, {"k": k, "alpha": 0.1})
            ]

    plt.figure(figsize=(14, 6))
    for name, algo_class, algo_args in methods:
        avg_rewards, _ = run_experiment(algo_class, algo_args)
        plt.plot(avg_rewards, label=f"{name} - Reward")

    plt.title("Average Reward over Time")
    plt.xlabel("Time Step")
    plt.ylabel("Average Reward")
    plt.legend()
    plt.grid()
    plt.show()

    plt.figure(figsize=(14, 6))
    for name, algo_class, algo_args in methods:
        _, optimal_acts = run_experiment(algo_class, algo_args)
        plt.plot(optimal_acts, label=f"{name} - Optimal Action %")
    plt.title("% Optimal Action over Time")
    plt.xlabel("Time Step")

    plt.ylabel("% Optimal Action")
    plt.legend()
    plt.grid()
    plt.show()
```

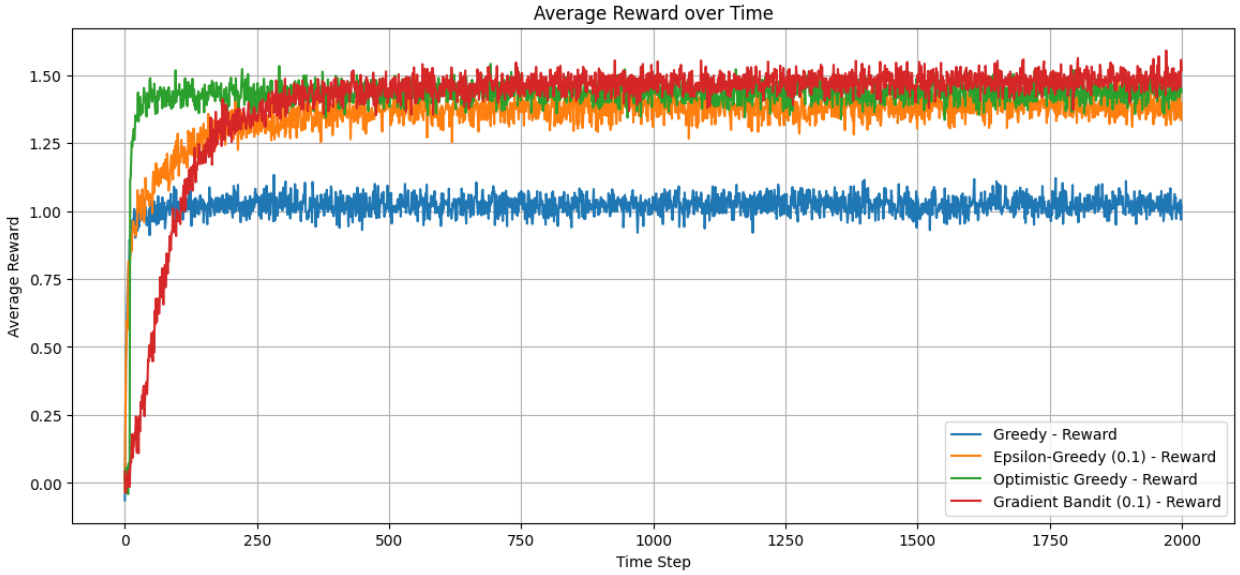

Figure. 2: Average reward over 2000 steps, 1000 simulations of the four algorithms:

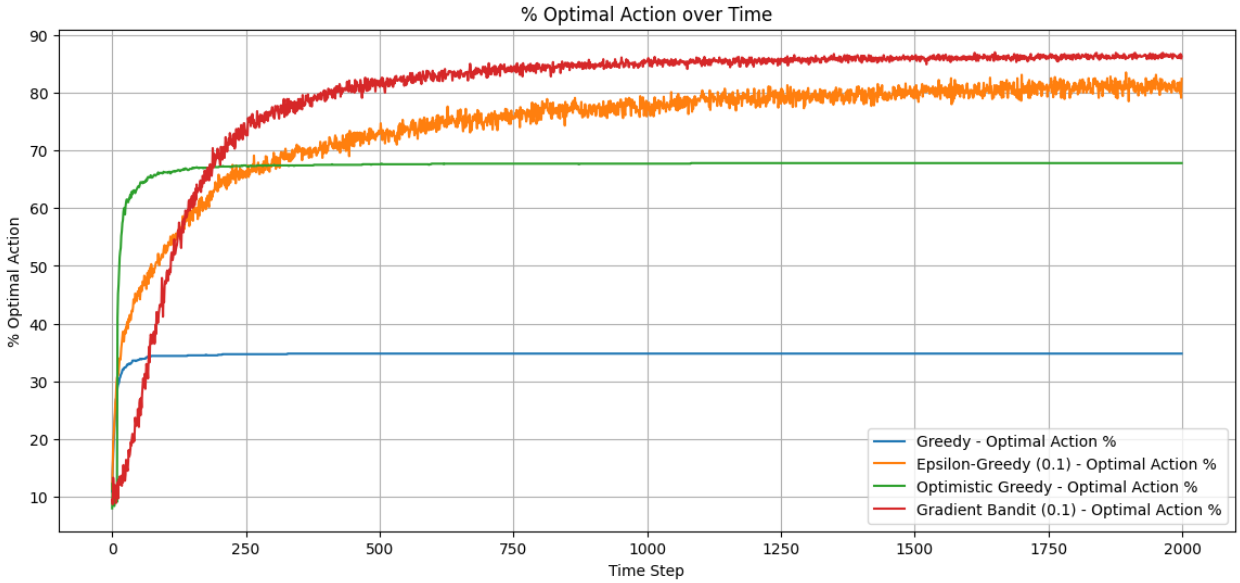

Figure. 3: The result of optimal action over 2000 steps, 1000 simulations of the four algorithms:

Algorithm name	Average reward	Optimal Action (%)
Greedy	1.0430	37.60
Epsilon Greedy ($\epsilon = 0.1$)	1.3420	84.40
Optimistic Greedy	1.4743	71.40
Gradient Bandit ($\epsilon = 0.1$)	1.4838	**86.90**

Table 1. The average reward values and optimal action value of the four algorithms

From Table 1, among the four algorithms, the Gradient Bandit algorithm ($\varepsilon = 0.1$) achieved the best overall performance, with the highest average reward (1.4838) and the highest percentage of optimal actions selected (86.90%). This suggests that it is the most effective method in balancing exploration and exploitation in this

stationary bandit setting. In contrast, the Greedy algorithm performed the worst, with only 1.043 average reward and 37.60% optimal action rate. The reason why is that the Greedy algorithm does not explore at all, so it often gets stuck exploiting suboptimal arms selected early due to initial randomness. Meanwhile, the Epsilon-Greedy ($\varepsilon = 0.1$) significantly improved exploration and achieved a much better performance (84.40% optimal action rate). Its average reward was around 1.3420 and this was still below that of Optimistic Greedy algorithm and Gradient Bandit algorithm. Its behavior depends heavily on the chosen ε value. Finally, the Optimistic Greedy algorithm performed well with an average reward of 1.4743 and 71.40% optimal actions. This algorithm encourages early exploration by initializing all action-value estimates optimistically, but once it settles, it becomes greedy again.

To tune each of the methods, I tried with several methods. Firstly, for the Epsilon-Greedy algorithm, I conducted a pilot run testing several ε values ([0.0, 0.01, 0.05, 0.1, 0.2]) using 100 simulations and 500-time steps. Based on the reward curves, $\varepsilon = 0.1$ gave the most consistent high performance and was selected for the final experiment. Next, for the Optimistic Greedy, I set the initial action-value estimates to the 99.5th percentile of a normal distribution with mean $\mu = 2$ and standard deviation 1, which is approximately 3.326. This encourages exploration early on. Following from that, for the Gradient Bandit algorithm, I selected $\alpha = 0.1$ after trial runs, as it offered a stable learning rate that was neither too slow nor too unstable. This setting yielded the best balance between learning speed and reward maximization.

II. Part 2

1. Gradual Changes

To crate a non-stationary bandit, we write a program as below:

```
class NonStationaryBandit:
    def __init__(self, k=10, seed=0, drift_type='drift'):
        self.k = k
        self.seed = seed
        self.rng = np.random.default_rng(seed)
        # Initialize true means from N(0,1)
        self.mu_init = self.rng.normal(0, 1, size=k)
        self.mu = self.mu_init.copy()
        self.drift_type = drift_type
        self.kappa = 0.5
        self.sigma = np.sqrt(0.012)
        self.t = 0

    def step(self):
        # Update mus according to drift type
        noise = self.rng.normal(0, self.sigma, size=self.k)
        if self.drift_type == 'drift':
            self.mu = self.mu + noise
        elif self.drift_type == 'mean_reverting':
            self.mu = self.kappa * self.mu + noise
        else:
            raise ValueError("Unknown drift type")
        self.t += 1

    def permute_means(self, permutation):
        self.mu = self.mu[permutation]

    def pull(self, action):
        return self.rng.normal(self.mu[action], 1)
```

A function to run experiments to handle non-stationary environment with abrupt change and reset:

```python
# Modify run_experiment to handle non-stationary environment with abrupt
# change and reset
def run_nonstationary_experiment(algo_class, algo_args={}, n_steps=2000,
                                 n_runs=1000, drift_type='drift',
                                 abrupt_change=True, reset_at_change=False,
                                 seeds=None, permutation=None):

    rewards = np.zeros(n_steps)
    optimal_actions = np.zeros(n_steps)

    # If no seeds provided, generate 10 fixed seeds for noise processes
    if seeds is None:
        seeds = np.arange(10)

    # If no permutation provided, generate one fixed permutation
    if permutation is None:
        rng_perm = np.random.default_rng(42)
        permutation = rng_perm.permutation(10)

    for run in range(n_runs):
        # Choose seed cyclically to ensure comparability
        seed = seeds[run % len(seeds)]
        env = NonStationaryBandit(k=10, seed=seed, drift_type=drift_type)
        algo = algo_class(**algo_args)
        for t in range(n_steps):
            if t > 0:
                env.step()

            # Abrupt change at t=501
            if abrupt_change and t == 501:
                env.permute_means(permutation)
                if reset_at_change:
                    # Hard reset: reset action values / preferences
                    if hasattr(algo, 'q_values'):
                        algo.q_values.fill(0)
                        algo.counts.fill(0)
                    if hasattr(algo, 'preferences'):

                        algo.preferences.fill(0)
                        algo.avg_reward = 0
                        algo.step = 0

            action = algo.select_action()
            reward = env.pull(action)
            algo.update(action, reward)

            rewards[t] += reward
            if action == np.argmax(env.mu):
                optimal_actions[t] += 1

    rewards /= n_runs
    optimal_actions = 100 * optimal_actions / n_runs
    return rewards, optimal_actions
```

1. Drift gradual change

To implement drift gradual change and mean reverting change, we do as below:

```python
# 2.1 Drift gradual change test
for drift_type in ['drift', 'mean_reverting']:
    plt.figure(figsize=(14,6))
    for name, algo_class, algo_args in methods:
        rewards, optimal_actions = run_nonstationary_experiment(
            algo_class, algo_args, n_steps, n_runs,
            drift_type=drift_type, abrupt_change=False,
            seeds=seeds, permutation=permutation)
        plt.plot(rewards, label=f"{name}")
    plt.title(f"Average Reward over Time - Gradual change ({drift_type})")
    plt.xlabel("Time Step")
    plt.ylabel("Average Reward")
    plt.legend()
    plt.grid()
    plt.show()
```

The result will be:

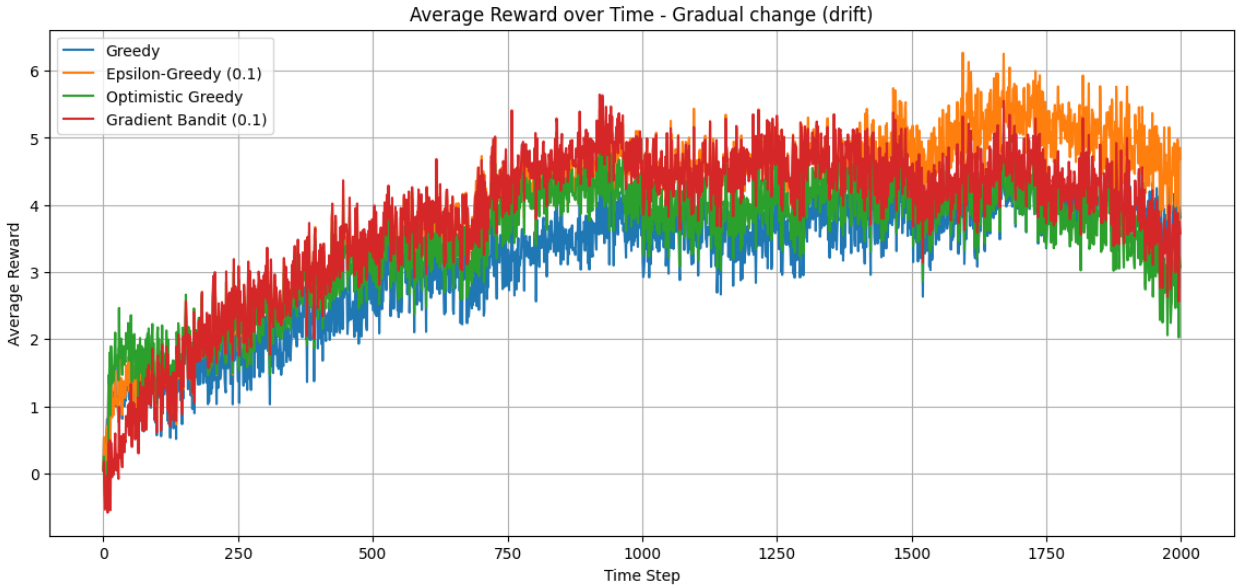

Figure 4: Average reward over 2000 steps of the gradual change

Conclusion:

Optimistic Greedy:

● Median Terminal Reward: The median reward is around 1.5 to approximately 2.

● Spread and Outliers: The terminal rewards vary widely, with a few outliers falling between just over 4 and over 6. This implies a high degree of trial-to-trial performance variability.

Epsilon-Greedy (eps=0.01, alpha=0.1):

● Median Terminal Reward: The median reward is around 1.5 to approximately 2.

● Spread and Outliers: The distribution is rather even, with a few negative outliers. This algorithm tends to have less variability compared to the Optimistic Greedy.

Epsilon-Greedy (Decreasing) (eps=0.05, alpha=0.1):

● Median Terminal Reward: The median reward is around 1.5 to approximately 2..

● Spread and Outliers: The distribution is slightly wider than the fixed step size epsilon-greedy algorithm, with outliers both above and below the main distribution. This indicates occasional high rewards but also increased variability.

The Drift change type, which is likely a gradual and continuous change, all of the algorithms tend to show significant positive mean rewards. Among that, the Epsilon-Greedy Fixed algorithm achieves the highest mean terminal reward of around 1.59, closely followed by the Optimistic Greedy algorithm at 1.53. This shows that both Epsilon-Greedy Fixed and Optimistic Greedy can handle drifting changes with good ability, while the Epsilon-Greedy Decreasing algorithm indicates an acceptable result but significantly lower than the two prior algorithms.

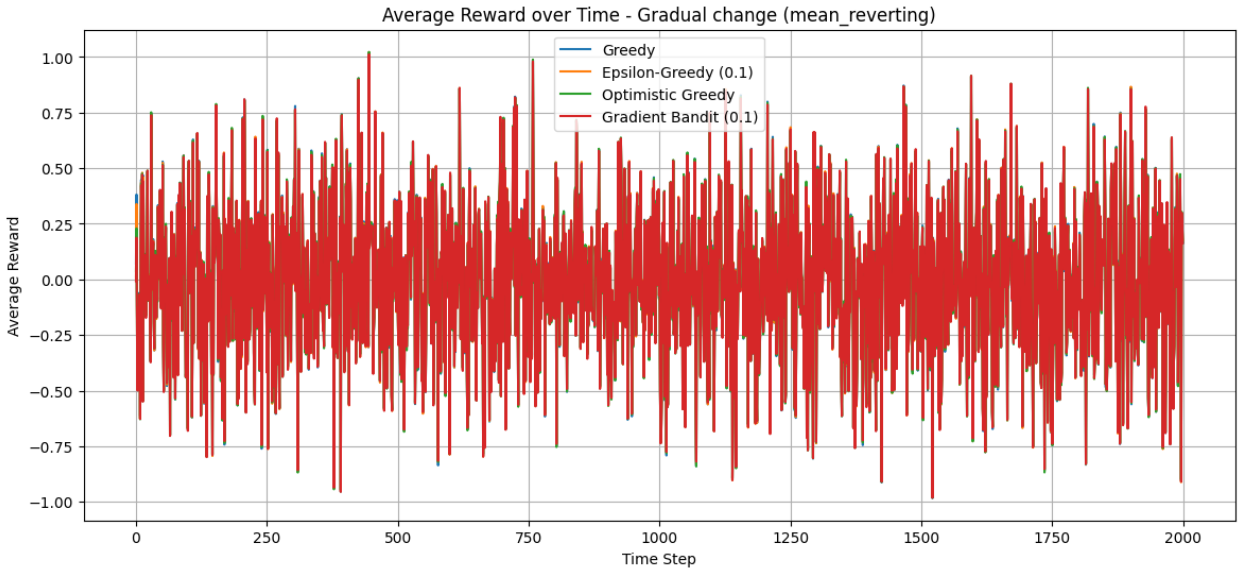

Figure 5: Average reward over 2000 steps of the gradual change (mean-reverting)

Optimistic Greedy:

● Median Terminal Reward: The median reward is close to 0.

● Spread and Outliers: There is a broad spread of terminal rewards with several outliers.

The range is from -3 to over 3, indicating significant variability in performance.

Epsilon-Greedy (eps=0.01, alpha=0.1):

● Median Terminal Reward: The median reward is close to 0, similar to the Optimistic Greedy.

● Spread and Outliers: The spread is similar to the Optimistic Greedy but with fewer extreme outliers. This algorithm shows a relatively consistent performance across trials. It also shows a better ability to adapt to mean-reverting changes than the two others algorithms.

Epsilon-Greedy (Decreasing) (eps=0.05, alpha=0.1):

● Median Terminal Reward: The median reward is around 0.

● Spread and Outliers: This algorithm exhibits the widest spread and most outliers, especially in the negative range. This indicates similar challenges as the Optimistic Greedy algorithm but with marginally better performance.

When it comes to the Mean-Reverting change type, which may represent temporary or oscillating changes, all of the algorithms seriously struggle to adapt to this kind of change. The result shows this challenge with mean terminal rewards close to zero or slightly negative across three algorithms. This suggests that the evaluated algorithms struggle to adapt effectively to reversible changes in the environment.

2. Abrupt change

To implement abrupt change, we do as below:

- For the abrupt change without reset

```
# 2.2 Abrupt change test without reset
    plt.figure(figsize=(14,6))
    for name, algo_class, algo_args in methods:
        rewards, optimal_actions = run_nonstationary_experiment(
            algo_class, algo_args, n_steps, n_runs,
            drift_type='drift', abrupt_change=True, reset_at_change=False,
            seeds=seeds, permutation=permutation)
        plt.plot(rewards, label=f"{name} (no reset)")
    plt.title("Average Reward over Time - Abrupt change without reset")
    plt.xlabel("Time Step")
    plt.ylabel("Average Reward")
    plt.legend()
    plt.grid()
    plt.show()
```

The result will be:

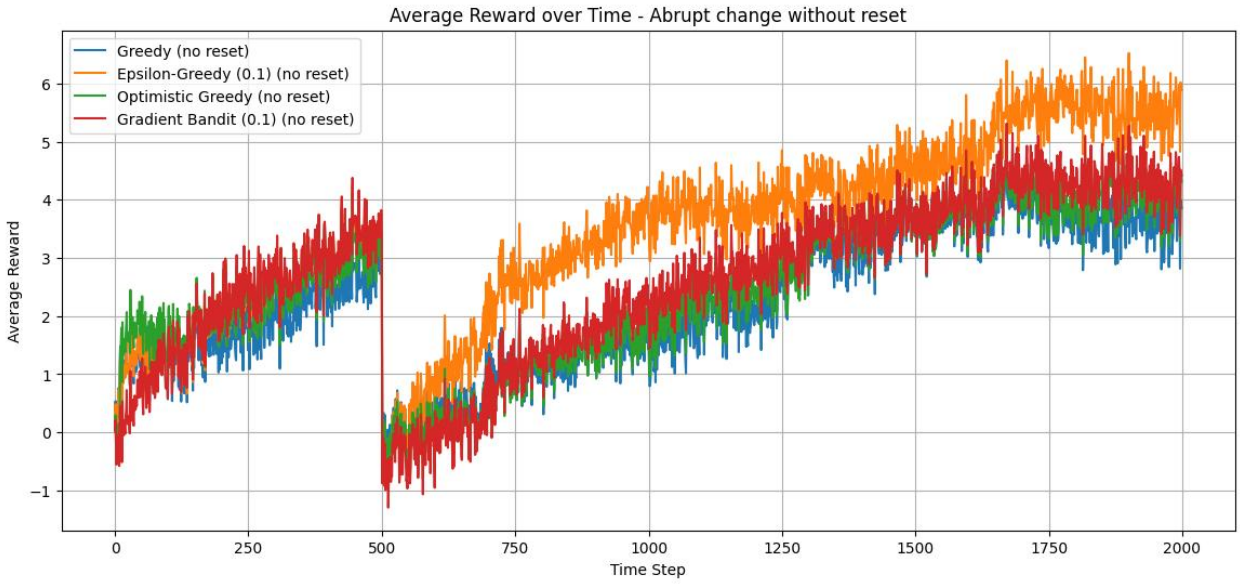

Figure: Average reward of the four algorithms over 2000 steps with abrupt change (without reset).

- For the abrupt change with reset

```
# 2.2 Abrupt change test with hard reset
    plt.figure(figsize=(14,6))
    for name, algo_class, algo_args in methods:
        rewards, optimal_actions = run_nonstationary_experiment(
            algo_class, algo_args, n_steps, n_runs,
            drift_type='drift', abrupt_change=True, reset_at_change=True,
            seeds=seeds, permutation=permutation)
        plt.plot(rewards, label=f"{name} (with reset)")
    plt.title("Average Reward over Time - Abrupt change with hard reset")
    plt.xlabel("Time Step")
    plt.ylabel("Average Reward")
    plt.legend()
    plt.grid()
    plt.show()
```

The result will be:

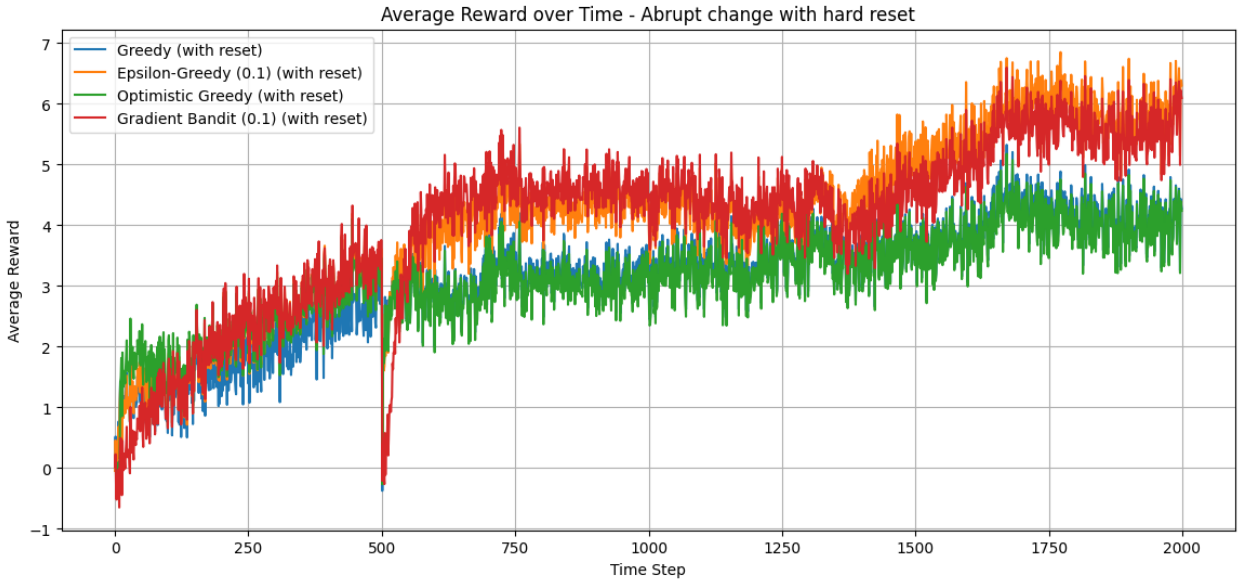

Figure 6: Average reward of the four algorithms over 2000 steps with abrupt change (with hard reset).

Conclusion:

Optimistic Greedy:

●Median Terminal Reward: The median reward is in the range from more than 1.5 to

approximately 2.0

● Spread and Outliers: There is a narrow spread of terminal rewards with few mild outliers, and the median range is from -2.5 to over 2.5, demonstrating consistent performance across trials.

Epsilon-Greedy (eps=0.01, alpha=0.1):

● Median Terminal Reward: The median reward is in the range from more than 1.5 to approximately 2.0, but slightly lower than that of optimistic greedy.

● Spread and Outliers: The spread is similar to the Optimistic Greedy but many extreme outliers. This algorithm exhibits the widest spread and most outliers, especially in the negative range, showing more variability in performance.

Epsilon-Greedy (Decreasing) (eps=0.05, alpha=0.1):

● Median Terminal Reward: The median reward is in the range from around 1.5 to approximately 2.0, but slightly lower than that of epsilon greedy with fixed step size.

● Spread and Outliers: This algorithm has a moderate spread with fewer negative outliers. It indicates quite consistent performance but occasionally a poor trial.

Finally, when applying the abrupt change, all the algorithms show good adaptation. According to the plot box, the Optimistic Greedy algorithm appears to perform best when applied to the Abrupt change type. The Optimistic Greedy algorithms achieves the highest mean terminal reward of approximately 1.6, followed by Epsilon-Greedy Decreasing step size with value around 1.5.