Project 1: Exploring Bandit Algorithms in Stationary and Non-Stationary Environments

Student:	Harshani Rathnayake
Departments:	Mathematics and Statistics and Computer Science
Date:	June 15, 2025
GitHub Repository:	`https://github.com/Harshani-Rathnayake/Project-1-Bandit-Algorithms.git`

Abstract

The multi-armed bandit (MAB) problem balances the competing objectives of exploration and exploitation by offering a straightforward framework for sequential decision-making under uncertainty. Several standard bandit algorithms, including the gradient bandit method, epsilon-greedy with tuned parameters, greedy algorithm, and optimistic initialization, are implemented and compared in this study under both stationary and non-stationary reward settings. The non-stationary environments are made to mimic the instability of reward systems in the actual world by incorporating mean-reverting dynamics, sudden changes in points, and smooth drift. Two main performance indicators are used to evaluate each approach over 1000 simulations: the percentage of optimal action selection and the average per-step reward. The results offer useful suggestions for algorithm selection in dynamic environments and empirical insight into the adaptability and learning capability of each method. The results and code are all made openly accessible and completely replicable.

Introduction

In artificial intelligence and reinforcement learning, sequential decision-making in unpredictable environments is a basic challenge. For investigating these problems, the multi-armed bandit (MAB) problem provides a standard framework. In its most basic form, an agent periodically chooses from a group of actions (or arms), each of which produces random rewards derived from unidentified distributions. The objective is to maximize cumulative reward over time, which calls for striking a balance between *exploitation*—using the existing knowledge to choose the most well-known action—and *exploration*—learning about lesser-known actions.[3]

In this study, several well-known multi-armed bandit algorithms are practically studied in various environmental settings. To be more precise, this paper compares:

- Greedy algorithms initialized with zero action values,
- Epsilon-greedy algorithms with exploration rates selected through pilot tuning,
- Optimistic initialization methods using domain knowledge,
- Gradient bandit algorithms with varying learning rates.

First, these algorithms are tested in a stationary environment with an initial reward distribution for each arm. The analysis is then extended to non-stationary environments, which are defined by either gradual drift, mean-reverting dynamics, or abrupt changes (changepoints). These hypothetical situations simulate real-life circumstances in which reward systems change over time as a result of shifting consumer habits, changes in the market, or environmental influences.[1]

Across 1000 independent simulations of 2000 time steps, the performance of each algorithm is based on the average per-step reward and the proportion of optimal actions selected. This research contributes to a better understanding of instructional techniques in practice by illuminating the relative advantages and disadvantages of each strategy in both stable and dynamic situations.

1 Part 1: Stationary Bandit Environment

This experiment uses the conventional k-armed bandit testbed with $k = 10$ to investigate and evaluate the performance of different classical bandit learning algorithms in a stationary environment. The true action-value $\mu_i \sim \mathcal{N}(0, 1)$ is calculated once every simulation and held constant in the reward distribution $\mathcal{N}(\mu_i, 1)$ that corresponds to each arm $i \in \{1, \ldots, 10\}$. The learning algorithms are evaluated as follows:

- **Greedy (Q=0):** Pure exploitation with action-value estimates initialized at zero.
- **Epsilon-Greedy:** Selects a random action with probability ε, otherwise selects the greedy action. The best ε was tuned through pilot experiments.
- **Optimistic Greedy:** Greedy approach with optimistic initial values set to the 99.5th percentile of the best arm's reward distribution to encourage exploration.
- **Gradient Bandit:** Policy-gradient method that uses preferences instead of value estimates and applies a softmax to select actions. The learning rate α was tuned experimentally.

Hyperparameter Tuning: Several choices of $\varepsilon \in \{0.01, 0.05, 0.1, 0.2\}$ were tested in pilot trials, and the value that produced the largest average reward in the last 100 steps across 200 simulations was selected. The gradient bandit technique was evaluated using a similar grid search strategy, examining $\alpha \in \{0.01, 0.05, 0.1, 0.4\}$. The 99.5th percentile of the best arm distribution averaged over several trials was the optimistic initial value.

Every algorithm was executed in 2000 time steps and replicated over 1000 independent simulations using various arm values and random seeds. The performance metrics are as follows:

1. Average reward at each time step.
2. Percentage of times the optimal action (arm with highest μ_i) was selected.

1.1 Results – Stationary Setting

The average reward per step for each method is compared in Figure 1. Lack of exploration causes the greedy algorithm to plateau too soon. Both gradient bandit and epsilon-greedy bandit techniques balance exploration and exploitation more effectively, resulting in larger average rewards.

The percentage of times the optimal action was selected over time is displayed in Figure 2, demonstrating how rapidly and reliably each approach learns to favour the better arm.

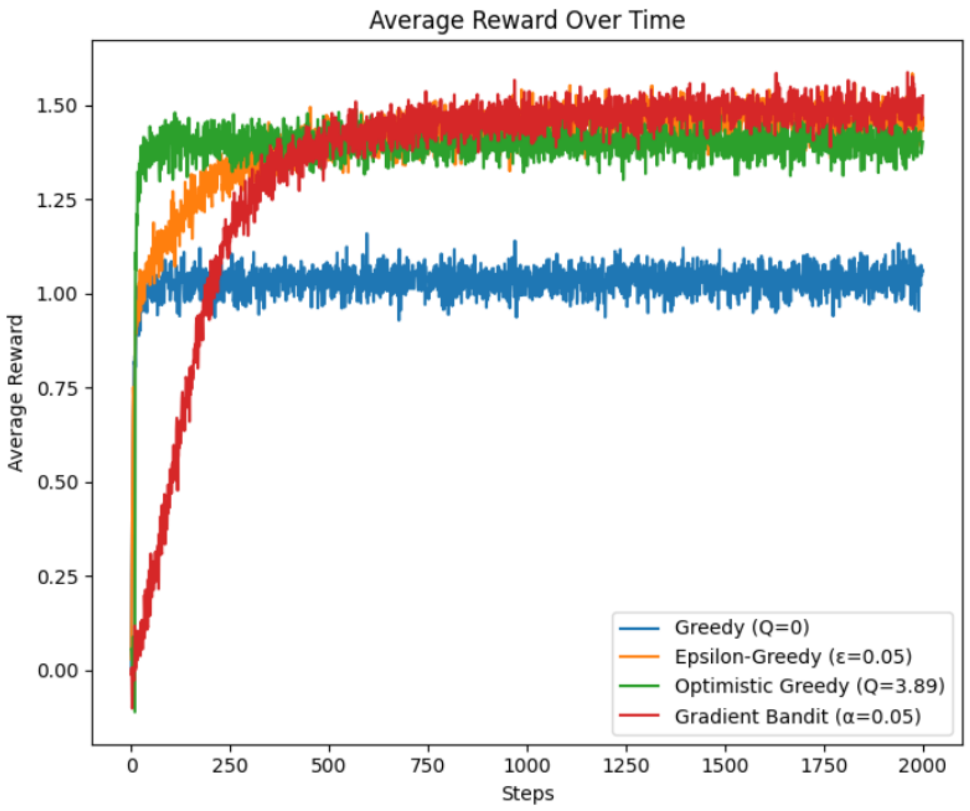

Figure 1: Average reward over 2000 steps for each algorithm.

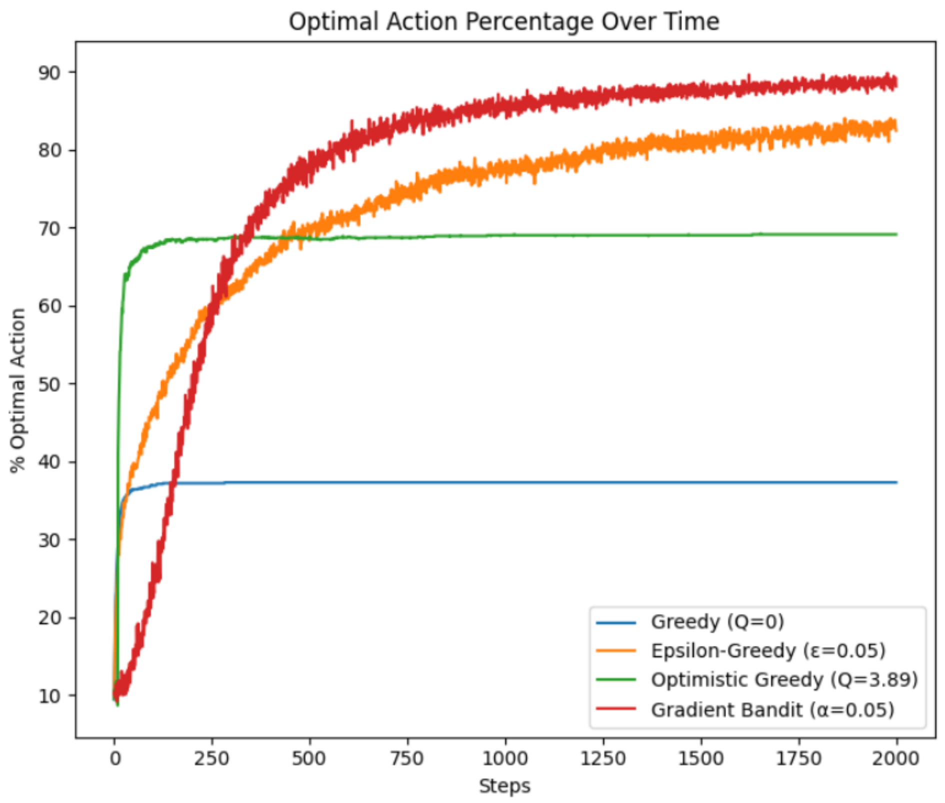

Figure 2: Percentage of time the optimal action is taken over 2000 steps for each algorithm.

Algorithm	**Final Average Reward**	**Percentage Optimal Action**
Greedy (Q=0)	1.043	37.30%
Epsilon-Greedy ($\varepsilon = 0.05$)	1.462	82.74%
Optimistic Greedy (Q=3.89)	1.399	69.10%
Gradient Bandit ($\alpha = 0.05$)	**1.493**	**88.58%**

Table 1: Performance metrics averaged over the final 100 steps.

1.2 Discussion – Part 1

The results make it extremely evident that exploration is essential to locating and taking advantage of the best action in stationary bandit environments. The low optimal action percentage and average reward show that the pure greedy algorithm, which lacks exploration, prematurely converges to ineffective actions.

The exploration and exploitation are successfully balanced by the epsilon-greedy strategy with $\varepsilon = 0.05$, which produces noticeably better results than greedy.

The optimistic greedy method promotes initial exploration with high starting values, but eventually behaves greedily, resulting in average performance.

Finally, the gradient bandit algorithm outperforms all others, likely due to its use of softmax action preferences, which allow it to continuously adapt based on reward feedback.

All things considered, the gradient bandit and epsilon-greedy algorithms perform better in stationary situations, with the gradient bandit obtaining the largest average reward and the best rate of action selection.

2 Part 2: Non-Stationary Bandit Environment

The reward distributions are frequently dynamic rather than constant in real-world situations. The multi-armed bandit problem is extended in this section to non-stationary environments, where the predicted rewards of actions change either suddenly or gradually. Under these circumstances, this study set several bandit algorithms into practice and evaluated them, assessing their adaptability by calculating average rewards and the percentage of times the most effective option of action is selected. [2]

2.1 Gradual Changes

The variations in the reward distributions over time replicate settings in which the underlying parameters gradually drift or return to a mean. Two categories of gradual changes were examined:

- **Drift:** The mean reward $\mu_{i,t}$ for each action is transformed into a random walk with Gaussian noise introduced, as modeled by

$$\mu_{i,t} = \mu_{i,t-1} + \epsilon_{i,t}, \quad \epsilon_{i,t} \sim \mathcal{N}(0, 0.0001)$$

- **Mean-Reverting:** The mean reward follows a mean-reverting process:

$$\mu_{i,t} = \kappa\mu_{i,t-1} + \epsilon_{i,t}, \quad \epsilon_{i,t} \sim \mathcal{N}(0, 0.0001)$$

  with $\kappa = 0.5$, causing the means to fluctuate around zero with added noise.

These small changes simulate things like changing brightness over the course of a day or gradually shifting user preferences.

The average reward and optimal action percentage for each algorithm under the drift scenario are plotted side by side in the following Figure 3 below.

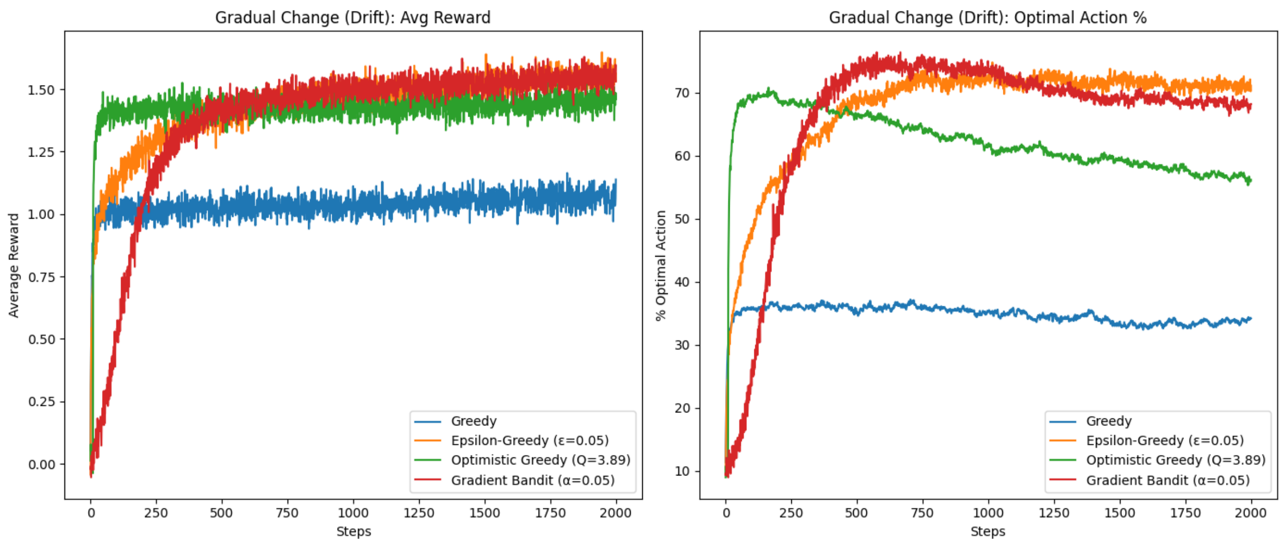

Figure 3: Gradual Change (Drift): Average Reward and Optimal Action Percentage for Different Bandit Algorithms

Comparable outcomes for the mean-reverting procedure are displayed in Figure 4 below.

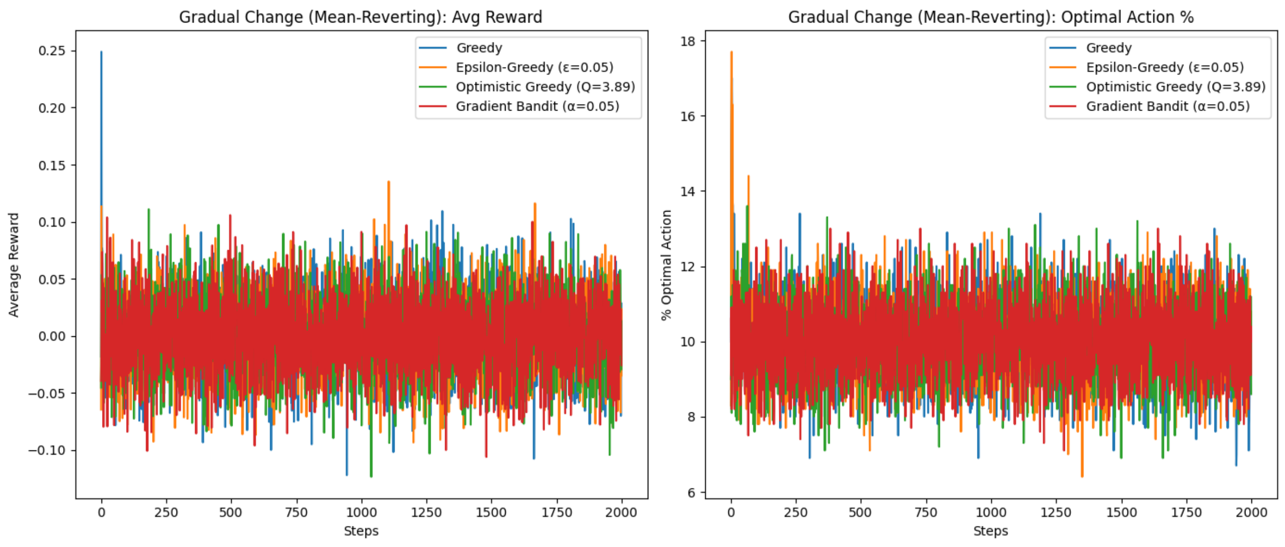

Figure 4: Gradual Change (Mean-Reverting): Average Reward and Optimal Action Percentage for Different Bandit Algorithms

2.2 Abrupt changes

Abrupt changes simulate rapid shifts, like turning a switch, in reward distribution. The reward means are randomly switched between acts at step 501, signifying an unanticipated shift in the surroundings. The two examples were investigated:

- **No Reset:** The bandit algorithms use their update mechanisms to adapt, continuing without resetting their learned values.

- **With Reset:** To simulate a situation in which the agent is aware of the change and begins learning again from scratch, the algorithms are "hard reset" at step 501 by setting all action-value estimates back to zero.

The two sudden change scenarios are shown below side by side with average reward and optimal action percentage graphs in Figures 5 and 6.

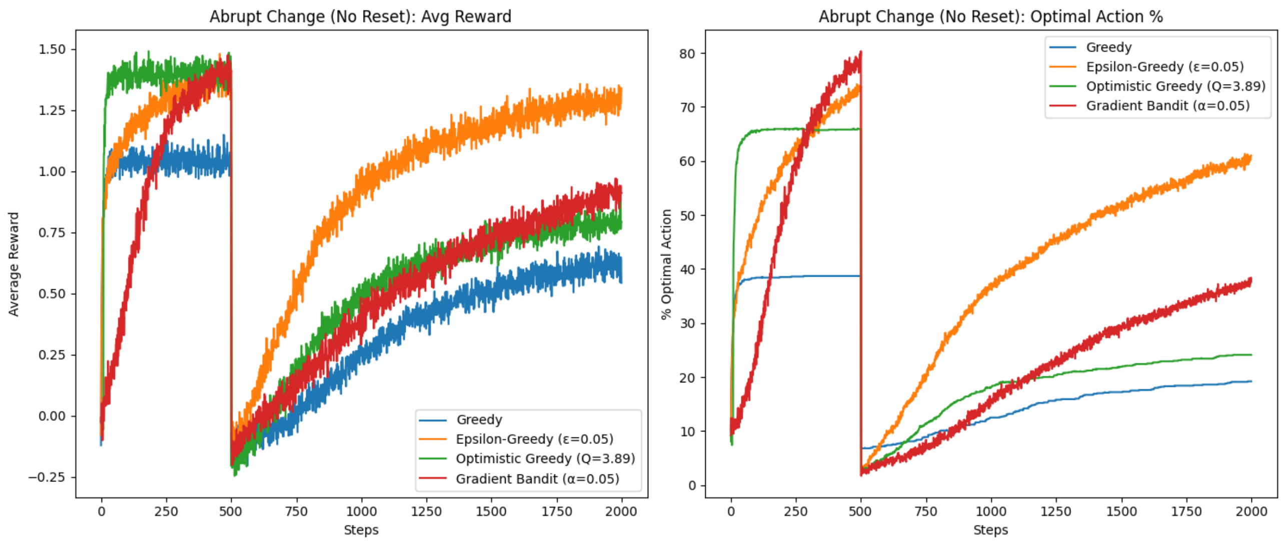

Figure 5: Abrupt Change (No Reset): Average Reward and Optimal Action Percentage for Different Bandit Algorithms

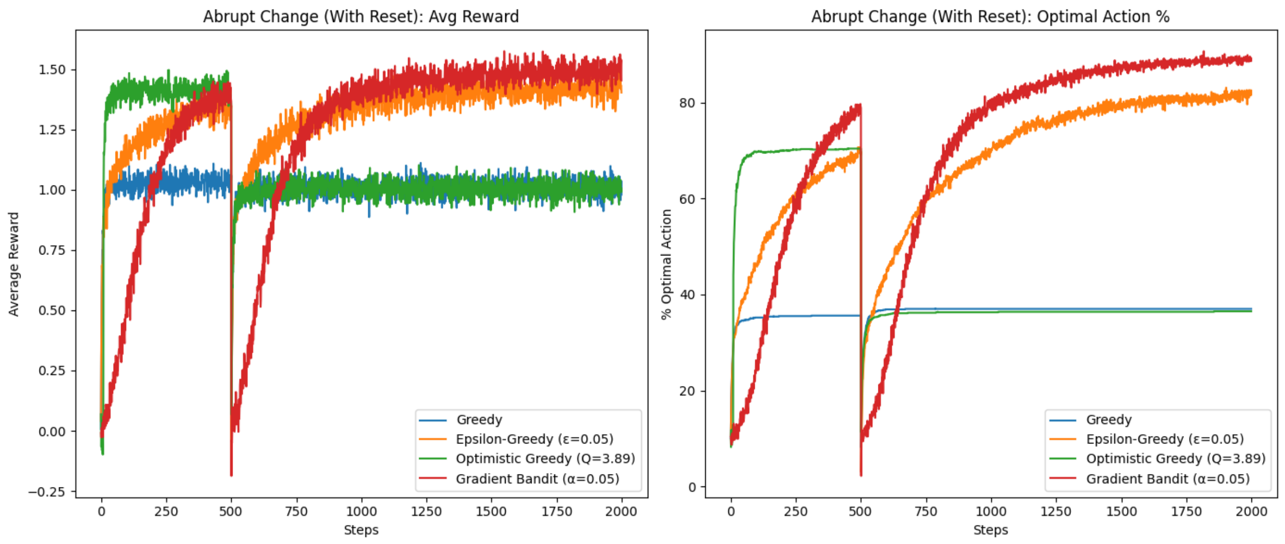

Figure 6: Abrupt Change (With Reset): Average Reward and Optimal Action Percentage for Different Bandit Algorithms

2.3 Results – Non-Stationary Setting

The performance reports for 2000 steps and 1000 runs provide numerous important insights:

Change Type	Algorithm	Average Reward	Percentage Optimal Action
Gradual Change (Drift)	Greedy	1.138	34.20%
	Epsilon-Greedy ($\epsilon = 0.05$)	1.563	71.10%
	Optimistic Greedy ($Q = 3.89$)	1.460	56.20%
	Gradient Bandit ($\alpha = 0.05$)	1.531	68.20%
Gradual Change (Mean-Reverting)	Greedy	0.028	9.60%
	Epsilon-Greedy ($\epsilon = 0.05$)	-0.035	9.90%
	Optimistic Greedy ($Q = 3.89$)	-0.029	8.60%
	Gradient Bandit ($\alpha = 0.05$)	-0.026	10.00%
Abrupt Change (No Reset)	Greedy	0.543	19.20%
	Epsilon-Greedy ($\epsilon = 0.05$)	1.339	61.00%
	Optimistic Greedy ($Q = 3.89$)	0.793	24.10%
	Gradient Bandit ($\alpha = 0.05$)	0.910	37.60%
Abrupt Change (With Reset)	Greedy	0.979	37.00%
	Epsilon-Greedy ($\epsilon = 0.05$)	1.402	82.50%
	Optimistic Greedy ($Q = 3.89$)	1.005	36.50%
	Gradient Bandit ($\alpha = 0.05$)	1.453	88.70%

Table 2: Final Summary of Bandit Algorithms Performance

- **Gradual Change (Drift):** The Epsilon-Greedy algorithm ($\epsilon = 0.05$) achieved the highest final average reward (1.563) and the highest percentage of optimal action selection (71.10%). The Gradient Bandit ($\alpha = 0.05$) also performed strongly, with a final average reward of 1.531 and 68.20% optimal actions. Greedy performed worst due to its lack of exploration, with a final average reward of 1.138 and only 34.20% optimal action selection.
- **Gradual Change (Mean-Reverting):** All methods struggled, showing near-zero or slightly negative average rewards and low optimal action percentages (around 8.6% to 10%). The mean-reverting nature of the environment likely caused rapid fluctuations that made stable learning difficult.
- **Abrupt Change (No Reset):** Performance decreased notably across all methods, especially Greedy and Optimistic Greedy, which showed lower average rewards (0.543 and 0.793 respectively) and optimal action percentages (19.20% and 24.10%). Epsilon-Greedy and Gradient Bandit adapted better but still suffered from the abrupt environmental changes, achieving 1.339 and 0.910 average rewards with 61.00% and 37.60% optimal actions respectively.
- **Abrupt Change (With Reset):** Explicitly resetting estimates at the change point improved adaptability significantly. Gradient Bandit ($\alpha = 0.05$) and Epsilon-Greedy ($\epsilon = 0.05$) achieved the highest final average rewards (1.453 and 1.402) and optimal action percentages (88.70% and 82.50%), substantially outperforming other methods.

2.4 Discussion – Part 2

These tests highlight the importance adaptability in non-stationary settings. Strong performance was continuously demonstrated by the Gradient Bandit algorithm, particularly in the drift and abrupt with reset scenarios. This is probably because the method is gradient-based and updates better monitor fluctuating reward structures. Although the epsilon-greedy technique did an outstanding job of balancing exploration, gradient methods somewhat overtook it.

All algorithms faced difficulties in the mean-reverting scenario, which highlighted the limitations when rewards oscillate unpredictably around a mean.

Reset or change-detection techniques are beneficial because abrupt changes without reset penalize algorithms that mostly count on accumulated past knowledge. In real-world applications, agents are capable to forget old information when their surroundings abruptly change.

All things considered, this section of the study demonstrated how several bandit algorithms handle time-varying reward distributions, highlighting the usefulness of integrating adaptation and reset techniques in sequential decision-making in the real world.

Conclusion

In conclusion, this study compared multiple bandit algorithms in stationary and non-stationary situations. The results revealed that exploration is critical, because the greedy approach performed poorly, whereas epsilon-greedy and gradient bandit strategies achieved bigger rewards and better action choices.

In dynamic circumstances, the gradient bandit algorithm generally performed well, particularly during steady drifts and rapid changes. Mean-reverting environments were problematic for all techniques, emphasizing the challenge of unstable rewards.

Moreover, reset learning after rapid changes boosted performance dramatically, demonstrating the need for adaptation. Overall, gradient-based and exploration-aware approaches are better adapted to real-world scenarios with dynamic reward structures.

Appendix

Part 1: Stationary Bandit Python Code

```python
#Step 1: Bandit Environment
class Bandit:
    def __init__(self, k=10):
        #This initializes a k-armed bandit with true action values sampled from a normal distribution
        self.k = k  #number of arms
        self.means = np.random.normal(0, 1, k)  #true reward means for each arm
        self.optimal_action = np.argmax(self.means)  #index of the best arm (highest mean)
    def step(self, action):
        #This generates a stochastic reward for the selected action (arm)
        return np.random.normal(self.means[action], 1)  # reward is drawn from N(true_mean, 1)

def argmax_random_tie(values):
    #This returns the index of the maximum value in the array, breaking ties randomly
    max_val = np.max(values)
    candidates = np.flatnonzero(values == max_val)  #indices with max value
    return np.random.choice(candidates)


#Step 2: Action Selection Algorithms
def greedy(bandit, steps=2000):
    #Greedy strategy: always pick the current best-known action (Q)
    Q = np.zeros(bandit.k)  #estimated rewards
    N = np.zeros(bandit.k) #count of times each arm was selected
    rewards = []
    optimal_actions = []
    for t in range(steps):
        action = argmax_random_tie(Q)  #select best current action
        reward = bandit.step(action)
        N[action] += 1
        Q[action] += (reward - Q[action]) / N[action] #incremental update
        rewards.append(reward)
        optimal_actions.append(int(action == bandit.optimal_action))
    return rewards, optimal_actions

def epsilon_greedy(bandit, epsilon=0.1, steps=2000):
    #Epsilon-greedy strategy: with probability epsilon, explore; otherwise exploit
    Q = np.zeros(bandit.k)
    N = np.zeros(bandit.k)
    rewards = []
    optimal_actions = []
    for t in range(steps):
        if np.random.rand() < epsilon:
            action = np.random.randint(bandit.k)  #explore
        else:
            action = argmax_random_tie(Q)  #exploit
        reward = bandit.step(action)
        N[action] += 1
        Q[action] += (reward - Q[action]) / N[action]
        rewards.append(reward)
        optimal_actions.append(int(action == bandit.optimal_action))
    return rewards, optimal_actions

def optimistic_greedy(bandit, initial_value=5.0, steps=2000):
    #Optimistic greedy: initializes estimates with high values to encourage initial exploration
    Q = np.ones(bandit.k) * initial_value  #optimistic start
    N = np.zeros(bandit.k)
    rewards = []
    optimal_actions = []
    for t in range(steps):
        action = argmax_random_tie(Q)
        reward = bandit.step(action)
        N[action] += 1
        Q[action] += (reward - Q[action]) / N[action]
        rewards.append(reward)
        optimal_actions.append(int(action == bandit.optimal_action))
    return rewards, optimal_actions

def gradient_bandit(bandit, alpha=0.1, steps=2000):
    #Gradient bandit algorithm using preference values and softmax action selection
    H = np.zeros(bandit.k)  #preferences
    avg_reward = 0
    rewards = []
    optimal_actions = []
    for t in range(steps):
        probs = np.exp(H) / np.sum(np.exp(H))  #softmax over preferences
        action = np.random.choice(bandit.k, p=probs) #pick action based on probabilities
        reward = bandit.step(action)
        avg_reward += (reward - avg_reward) / (t + 1)  #update average reward
        #Update preferences:
        for a in range(bandit.k):
            H[a] += alpha * ((1 if a == action else 0) - probs[a]) * (reward - avg_reward)
        rewards.append(reward)
        optimal_actions.append(int(action == bandit.optimal_action))
    return rewards, optimal_actions

#Step 3: Run a Simulation
def run_simulation(algo_func, runs=1000, **kwargs):
    #Below runs a bandit algorithm multiple times and average the results
    steps = kwargs.get('steps', 2000)
    avg_rewards = np.zeros(steps)
    optimal_action_counts = np.zeros(steps)
    for _ in range(runs):
        bandit = Bandit()
        rewards, optimal_actions = algo_func(bandit, **kwargs)
        avg_rewards += rewards
        optimal_action_counts += optimal_actions
    avg_rewards /= runs
    optimal_action_counts = (optimal_action_counts / runs) * 100  #convert to percentage
    return avg_rewards, optimal_action_counts

#Step 4: Tune the Initial Value for Optimistic Greedy
optimistic_values = []
for _ in range(20):
    b = Bandit()
    max_mean = max(b.means)
    # Calculate a high quantile value from a normal distribution centered on
     max_mean
    percentile = np.percentile(np.random.normal(loc=max_mean, scale=1, size
    =100000), 99.5)
    optimistic_values.append(percentile)
#Below is the final average optimistic Q-value
optimistic_value = np.mean(optimistic_values)


#Step 5: Tune Epsilon
epsilons = [0.01, 0.05, 0.1, 0.2]
epsilon_results = {}
#Trying different epsilons and record final average rewards
for eps in epsilons:
    rewards, _ = run_simulation(epsilon_greedy, epsilon=eps, runs=200, steps
    =500)
    epsilon_results[eps] = np.mean(rewards[-100:])  #average of last 100
    steps
#Choose epsilon that gave highest average reward
best_eps = max(epsilon_results, key=epsilon_results.get)


#Step 6: Tune Alpha
alphas = [0.01, 0.05, 0.1, 0.4]
alpha_results = {}
#Tring different alphas and record final average rewards
for a in alphas:
    rewards, _ = run_simulation(gradient_bandit, alpha=a, runs=200, steps
    =500)
    alpha_results[a] = np.mean(rewards[-100:])

#Choose the best alpha value
best_alpha = max(alpha_results, key=alpha_results.get)

#Step 7: Final Run of the Algorithms
results = {
    "Greedy (Q=0)": run_simulation(greedy),
    f"Epsilon-Greedy (ϵ={best_eps})": run_simulation(epsilon_greedy, epsilon
    =best_eps),
    f"Optimistic Greedy (Q={optimistic_value:.2f})": run_simulation(
    optimistic_greedy, initial_value=optimistic_value),
    f"Gradient Bandit (α={best_alpha})": run_simulation(gradient_bandit,
    alpha=best_alpha)
}


#Step 8: Ploting the Results
def plot_results(results):
    plt.figure(figsize=(14, 6))
    #Plot average reward:
    plt.subplot(1, 2, 1)
    for label, (rewards, _) in results.items():
        plt.plot(rewards, label=label)
    plt.xlabel("Steps")
    plt.ylabel("Average Reward")
    plt.title("Average Reward Over Time")
    plt.legend()

    #Plot optimal action percentage:
```

```python
    plt.subplot(1, 2, 2)
    for label, (_, optimal_actions) in results.items():
        plt.plot(optimal_actions, label=label)
    plt.xlabel("Steps")
    plt.ylabel("% Optimal Action")
    plt.title("Optimal Action Percentage Over Time")
    plt.legend()
    plt.tight_layout()
    plt.show()
#Calling the plot function
plot_results(results)

#Step 9: Final Summary Output
#Below prints the final performance for each algorithm
for label, (rewards, optimal) in results.items():
    final_avg_reward = np.mean(rewards[-100:]) #last 100 steps
    final_opt_action = np.mean(optimal[-100:])  #last 100 steps
    print(f"{label}: Final Avg Reward = {final_avg_reward:.3f}, Final % Optimal = {final_opt_action:.2f}%")
```

Listing 1: Python Code for Stationary Bandit Algorithms

Part 2: Non-Stationary Bandit Python Code

```python
#Non-Stationary Bandit Class:
class NonStationaryBandit:
    def __init__(self, k=10, change_type=None, seed=None, permutation=None, noise_stream=None):
        self.k = k  #number of arms (actions)
        rng = np.random.default_rng(seed)  #reproducible random generator with optional seed
        self.means = rng.normal(0, 1, k)  #initialize mean reward for each arm from N(0,1)
        self.original_means = self.means.copy()  #backup of original means (for comparison or reset)
        self.optimal_action = np.argmax(self.means)  #initially optimal action (arm with highest mean)
        self.change_type = change_type  #type of environment change ('drift', 'abrupt', etc)
        self.permutation = permutation  #used in abrupt change (permutes the arm means)
        self.noise_stream = noise_stream  #stream of noise vectors for gradual changes

    def step(self, t, action):
        #Apply changes to the means based on the type of environment
        if self.change_type == 'drift':
            #Gradually drift the means with small noise
            if t > 0:
                self.means += self.noise_stream[t]
        elif self.change_type == 'mean_reverting':
            #Move means closer to zero with some noise
            if t > 0:
                self.means = 0.5 * self.means + self.noise_stream[t]
        elif self.change_type == 'abrupt' and t == 501:
            #At time t=501, apply abrupt permutation of means
            self.means = self.means[self.permutation]
        #Update optimal action after changes
        self.optimal_action = np.argmax(self.means)
        #Return a sample reward from N(mean[action], 1)
        return np.random.normal(self.means[action], 1)

#Runner for Non-Stationary Bandit Algorithms:
def run_nonstationary(algo_func, change_type, reset=False, **kwargs):
    runs = kwargs.get('runs', 1000)
    steps = kwargs.get('steps', 2000)
    avg_rewards = np.zeros(steps)  #to accumulate average rewards across all runs
    optimal_action_counts = np.zeros(steps)  #to accumulate % optimal actions
    seeds = np.arange(runs)
    #Create a fixed permutation for abrupt changes
    permutation = np.random.permutation(10)
    #Create noise streams for each run - used in drift or mean-reverting
    noise_streams = [np.random.normal(0, 0.012, (steps, 10)) for _ in range(runs)]
    #Run simulation over multiple independent seeds
    for seed in seeds:
        bandit = NonStationaryBandit(
            change_type=change_type,
            seed=seed,
            permutation=permutation,
            noise_stream=noise_streams[seed]
        )
        #Initialization of parameters
        Q = np.zeros(bandit.k)  #estimated value of each arm
        N = np.zeros(bandit.k) #number of times each arm was selected
        H = np.zeros(bandit.k) #preference values for gradient bandit
        avg_reward = 0  #running average of reward for gradient bandit
        #Extract tuning hyperparameters
        alpha = kwargs.get('alpha', 0.1)
        epsilon = kwargs.get('epsilon', 0.1)
        init_val = kwargs.get('initial_value', 0)
        #For optimistic method, set high initial Q values
        if algo_func.__name__ == "optimistic_greedy":
            Q = np.ones(bandit.k) * init_val
        rewards = []
        optimal_actions = []
        for t in range(steps):
            #Action selection based on algorithm type
            if algo_func.__name__ == "epsilon_greedy":
                if np.random.rand() < epsilon:
                    action = np.random.randint(bandit.k)
                else:
                    action = argmax_random_tie(Q)
            elif algo_func.__name__ in ["greedy", "optimistic_greedy"]:
                action = argmax_random_tie(Q)
            elif algo_func.__name__ == "gradient_bandit":
                probs = np.exp(H) / np.sum(np.exp(H))
                action = np.random.choice(bandit.k, p=probs)
            #Get reward from bandit
            reward = bandit.step(t, action)
            #Optional reset at time of abrupt change
            if reset and t == 501:
                Q = np.zeros(bandit.k)
                N = np.zeros(bandit.k)
                H = np.zeros(bandit.k)
                avg_reward = 0
            #Update estimates or preferences
            if algo_func.__name__ in ["greedy", "epsilon_greedy", "
```

```
   optimistic_greedy"]:
                N[action] += 1
                Q[action] += (reward - Q[action]) / N[action]
            elif algo_func.__name__ == "gradient_bandit":
                avg_reward += (reward - avg_reward) / (t + 1)
                for a in range(bandit.k):
                    H[a] += alpha * ((1 if a == action else 0) - probs[a]) *
    (reward - avg_reward)
            #Record metrics
            rewards.append(reward)
            optimal_actions.append(int(action == bandit.optimal_action))
        avg_rewards += rewards
        optimal_action_counts += optimal_actions
    #Average metrics across runs
    avg_rewards /= runs
    optimal_action_counts = (optimal_action_counts / runs) * 100
    return avg_rewards, optimal_action_counts


#Plotting Function for Non-Stationary Experiments:
def plot_nonstationary(results, title):
    plt.figure(figsize=(14, 6))
    #Plot: Average reward over Time
    plt.subplot(1, 2, 1)
    for label, (rewards, _) in results.items():
        plt.plot(rewards, label=label)
    plt.xlabel("Steps")
    plt.ylabel("Average Reward")
    plt.title(f"{title}: Avg Reward")
    plt.legend()
    #Plot: Optimal action percentage over Time
    plt.subplot(1, 2, 2)
    for label, (_, optimal_actions) in results.items():
        plt.plot(optimal_actions, label=label)
    plt.xlabel("Steps")
    plt.ylabel("% Optimal Action")
    plt.title(f"{title}: Optimal Action %")
    plt.legend()
    plt.tight_layout()
    plt.show()

#Run Experiments for Different Change Types:
#Gradual Drift Change (non-stationary rewards increase/decrease slowly):
drift_results = {
    "Greedy": run_nonstationary(greedy, change_type='drift'),
    f"Epsilon-Greedy (ϵ={best_eps})": run_nonstationary(epsilon_greedy,
   change_type='drift', epsilon=best_eps),
    f"Optimistic Greedy (Q={optimistic_value:.2f})": run_nonstationary(
   optimistic_greedy, change_type='drift', initial_value=optimistic_value),
    f"Gradient Bandit (α={best_alpha})": run_nonstationary(gradient_bandit,
   change_type='drift', alpha=best_alpha)
}
plot_nonstationary(drift_results, "Gradual Change (Drift)")

#Gradual Mean-Reverting Change (rewards decay towards zero):
reverting_results = {
    "Greedy": run_nonstationary(greedy, change_type='mean_reverting'),
    f"Epsilon-Greedy (ϵ={best_eps})": run_nonstationary(epsilon_greedy,
   change_type='mean_reverting', epsilon=best_eps),
    f"Optimistic Greedy (Q={optimistic_value:.2f})": run_nonstationary(
    optimistic_greedy, change_type='mean_reverting', initial_value=
    optimistic_value),
    f"Gradient Bandit (α={best_alpha})": run_nonstationary(gradient_bandit,
    change_type='mean_reverting', alpha=best_alpha)
}
plot_nonstationary(reverting_results, "Gradual Change (Mean-Reverting)")

#Abrupt Change WITHOUT Reset (agent continues using old Q/H):
abrupt_norestart_results = {
    "Greedy": run_nonstationary(greedy, change_type='abrupt', reset=False),
    f"Epsilon-Greedy (ε={best_eps})": run_nonstationary(epsilon_greedy,
    change_type='abrupt', epsilon=best_eps, reset=False),
    f"Optimistic Greedy (Q={optimistic_value:.2f})": run_nonstationary(
    optimistic_greedy, change_type='abrupt', initial_value=optimistic_value,
    reset=False),
    f"Gradient Bandit (α={best_alpha})": run_nonstationary(gradient_bandit,
    change_type='abrupt', alpha=best_alpha, reset=False)
}
plot_nonstationary(abrupt_norestart_results, "Abrupt Change (No Reset)")

#Abrupt Change WITH Reset (agent resets internal estimates at t=501):
abrupt_reset_results = {
    "Greedy": run_nonstationary(greedy, change_type='abrupt', reset=True),
    f"Epsilon-Greedy (ε={best_eps})": run_nonstationary(epsilon_greedy,
    change_type='abrupt', epsilon=best_eps, reset=True),
    f"Optimistic Greedy (Q={optimistic_value:.2f})": run_nonstationary(
    optimistic_greedy, change_type='abrupt', initial_value=optimistic_value,
    reset=True),
    f"Gradient Bandit (α={best_alpha})": run_nonstationary(gradient_bandit,
    change_type='abrupt', alpha=best_alpha, reset=True)
}
plot_nonstationary(abrupt_reset_results, "Abrupt Change (With Reset)")

#Final Summary of the values:
def print_final_summary(results, title):
    print(f"\n=== Final Summary for {title} ===")
    for label, (avg_rewards, optimal_actions) in results.items():
        final_avg_reward = avg_rewards[-1]
        final_optimal_pct = optimal_actions[-1]
        print(f"{label}: Final Avg Reward = {final_avg_reward:.3f}, Final %
    Optimal Action = {final_optimal_pct:.2f}%")
    print("="*40)
#Print summary of each experiment:
print_final_summary(drift_results, "Gradual Change (Drift)")
print_final_summary(reverting_results, "Gradual Change (Mean-Reverting)")
print_final_summary(abrupt_norestart_results, "Abrupt Change (No Reset)")
print_final_summary(abrupt_reset_results, "Abrupt Change (With Reset)")
```

Listing 2: Python Code for Non-Stationary Bandit Algorithms

References

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[2] Aurélien Garivier and Éric Moulines. Non-stationary bandit algorithms. In *Advances in Neural Information Processing Systems*, pages 1769–1776, 2008.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2nd edition, 2018.