Nicolas Stupak – 405178

Bandycki Streaming

1. Opis ćwiczenia

Celem laboratorium było przetestowanie różnych podejść do rozwiązywania problemu k-rękich bandytów. Nasz wariant tego problemu polegał na przewidywaniu, który z utworów może stać się przyszłym hitem. Nagroda 1 oznacza słuchacza, który przesłuchał go w całości, nagroda 0 oznacza, że utwór został pominięty. Do zaimplementowania były następujące algorytmy:

- Explore-Then Commit

- Greedy

- UCB1

- Gradient

- Thompson Sampling

2. Implementacja metod

1. Explore-Then Commit

```python
class ExploreThenCommitLearner(BanditLearner):
    def __init__(self, m: int = 10):
        self.name = "ETC"
        self.color = "blue"
        self.arms: list[str] = []
        self.time_step = 0
        self.k_arms: int = None
        self.m = m
        self.arms_stats: dict[str] = {}

    def reset(self, arms: list[str], time_steps: int):
        self.arms = arms
        self.k_arms = len(arms)
        self.arms_stats = {arm: 0 for arm in self.arms}
        self.time_step = 0

    def pick_arm(self) -> str:
        if self.time_step < self.m * self.k_arms:
            self.time_step += 1
            return self.arms[(self.time_step % self.k_arms)]
        else:
            return max(self.arms_stats, key=self.arms_stats.get)

    def acknowledge_reward(self, arm: str, reward: float) -> None:
        self.arms_stats[arm] += reward

```

2. Greedy

```python
class GreedyLearner(BanditLearner):
    def __init__(self, strategy=None, Q=None, epsilon=None):
        self.color = "purple"
        self.arms: list[str] = []
        self.time_step = 0
        self.k_arms: int = None
        self.arms_stats: dict[dict[int, int]] = {}

        if not strategy:
            self.strategy = random.choice(["Greedy", "ε-Greedy", "Optimistic-Greedy"])
        else:
            self.strategy = strategy

        # Initialize based on the chosen strategy
        if self.strategy == "Greedy":
            self.init_pure_greedy()
        elif self.strategy == "ε-Greedy":
            self.init_epsilon_greedy()
        else:
            self.init_optimistic_greedy()

    def init_pure_greedy(self):
        self.name = self.strategy

    def init_epsilon_greedy(self, epsilon=np.random.rand()):
        self.name = self.strategy
        self.epsilon = epsilon

    def init_optimistic_greedy(self, Q=random.randint(1, 11)):
        self.name = self.strategy
        self.Q = Q

    def reset(self, arms: list[str], time_steps: int):
        self.arms = arms
        self.k_arms = len(arms)
        self.arms_stats = {
            arm: {
                "n_pulls": 0,
                "mean": self.Q if self.strategy == "Optimistic-Greedy" else 0,
            }
            for arm in self.arms
        }
        self.time_step = 0

    def pick_arm(self) -> str:
        if self.time_step < self.k_arms:
            self.time_step += 1
            return self.arms[(self.time_step - 1)]
        else:
            if self.strategy != "ε-Greedy":
                return max(
                    self.arms_stats, key=lambda arm: self.arms_stats[arm]["mean"]
                )
            else:
                return np.random.choice(
                    [
                        random.choice(self.arms),
                        max(
                            self.arms_stats,
                            key=lambda arm: self.arms_stats[arm]["mean"],
```

```
                        ),
                    ],
                    p=[self.epsilon, 1 - self.epsilon],
                )

```

3. UCB1

```
class UpperConfidenceBoundLearner(BanditLearner):
    def __init__(self, c: int = 2):
        self.name = "UCB1"
        self.color = "red"
        self.arms: list[str] = []
        self.time_step = 0
        self.k_arms: int = None
        self.c = c
        self.arms_pulls: dict[str] = {}
        self.arms_rewards: dict[str] = {}
        self.arms_means: dict[str] = {}
        self.arms_ubc: dict[str] = {}

    def reset(self, arms: list[str], time_steps: int):
        self.arms = arms
        self.k_arms = len(arms)
        self.arms_pulls = {arm: 0 for arm in self.arms}
        self.arms_rewards = {arm: 0 for arm in self.arms}
        self.arms_means = {arm: 0 for arm in self.arms}
        self.arms_ubc = {arm: 0 for arm in self.arms}
        self.time_step = 0

    def pick_arm(self) -> str:
        self.time_step += 1
        if self.time_step <= self.k_arms:
            return self.arms[self.time_step - 1]
        else:
            self.calculate_ubc()
            return max(self.arms_ubc, key=self.arms_ubc.get)

    def acknowledge_reward(self, arm: str, reward: float) -> None:
        self.arms_rewards[arm] += reward
        self.arms_pulls[arm] += 1
        self.arms_means[arm] = self.arms_rewards[arm] / self.arms_pulls[arm]

    def calculate_ubc(self):
        for arm in self.arms:
            if self.arms_pulls[arm] == 0:
                self.arms_ubc[arm] = float("inf")
            else:
                self.arms_ubc[arm] = self.arms_means[arm] + (
                    self.c * np.sqrt(np.log(self.time_step) /
self.arms_pulls[arm])
                )

```

4. Gradient

```python
class GradientLearner(BanditLearner):
    def __init__(self, alpha: float = 0.1, random_alpha: bool = False):
        self.name = "Gradient"
        self.color = "cyan"
        self.alpha = np.random.rand() if random_alpha else alpha
        self.arms: list[str] = []
        self.time_step: int = 0
        self.reward_baseline: float = 0.0
        self.preferences: dict[str, float] = {}

    def reset(self, arms: list[str], time_steps: int):
        self.arms = arms
        self.preferences = {arm: 0.0 for arm in self.arms}
        self.time_step = 0
        self.reward_baseline = 0.0

    def pick_arm(self) -> str:
        probabilities = softmax(
            list(self.preferences.values())
        )  # Compute action probabilities
        return np.random.choice(self.arms, p=probabilities)

    def acknowledge_reward(self, arm: str, reward: float) -> None:
        """Update preferences based on the received reward"""
        self.time_step += 1
        self.reward_baseline += (1 / self.time_step) * (
            reward - self.reward_baseline
        )  # Update baseline

        probabilities = softmax(
            list(self.preferences.values())
        )  # Compute probabilities before updating

        for i, a in enumerate(self.arms):
            if a == arm:
                self.preferences[a] += (
                    self.alpha
                    * (reward - self.reward_baseline)
                    * (1 - probabilities[i])
                )
            else:
                self.preferences[a] -= (
                    self.alpha * (reward - self.reward_baseline) *
probabilities[i])

```

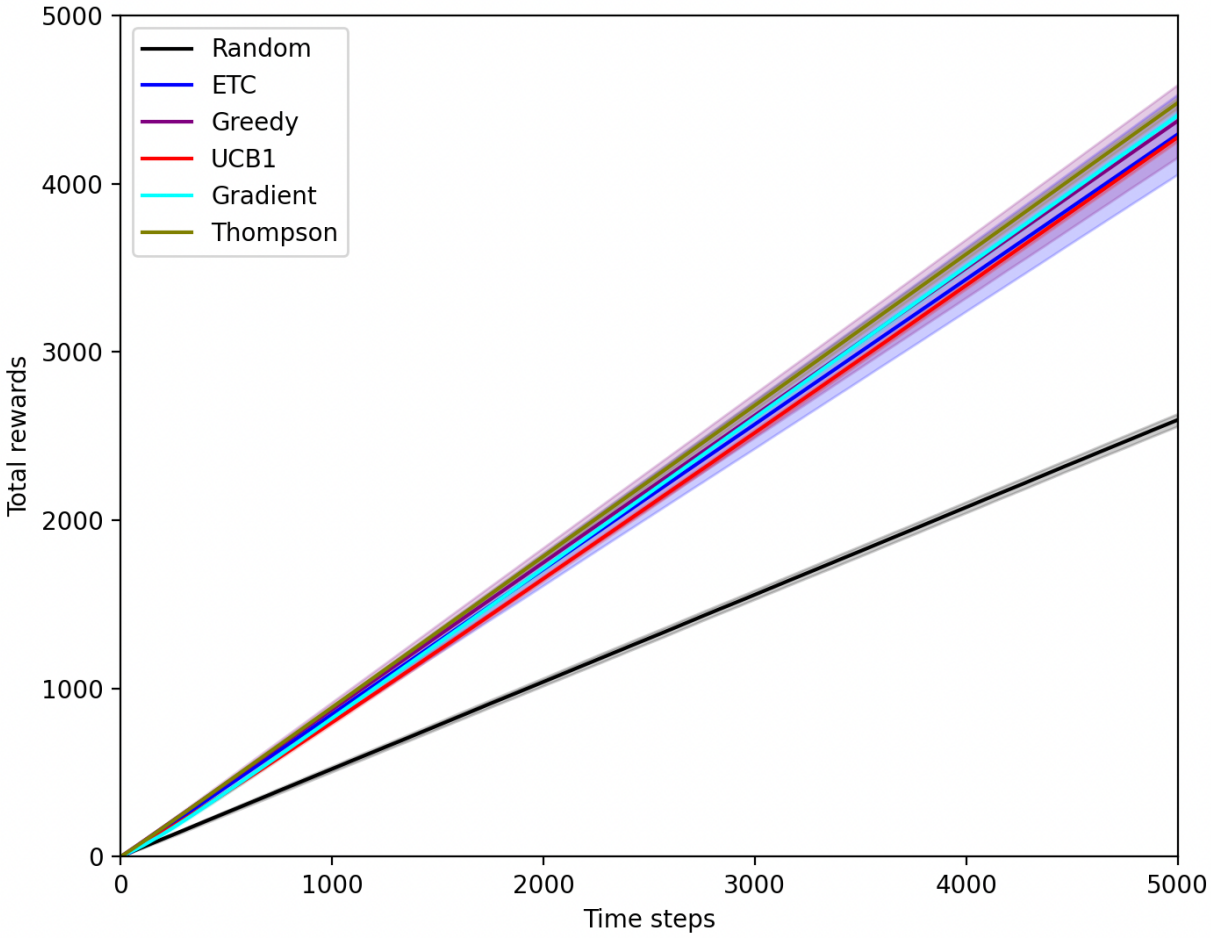
5. Thompson Sampling

```python
class ThompsonLearner(BanditLearner):
    def __init__(self):
        self.name = "Thompson"
        self.color = "olive"
        self.arms: list[str] = []
        self.time_step = 0
        self.k_arms: int = None
        self.arms_stats: dict[str] = {}

    def reset(self, arms: list[str], time_steps: int):
        self.arms = arms
```

```
        self.k_arms = len(arms)
        self.arms_stats = {arm: {"a": 1, "b": 1} for arm in self.arms}

    def pick_arm(self) -> str:
        return max(
            self.arms,
            key=lambda arm: beta.rvs(
                self.arms_stats[arm]["a"], self.arms_stats[arm]["b"]
            ),
        )

    def acknowledge_reward(self, arm: str, reward: float) -> None:
        self.arms_stats[arm]["a"] += reward
        self.arms_stats[arm]["b"] += 1 - reward

```

3. Realizacja eksperymentów

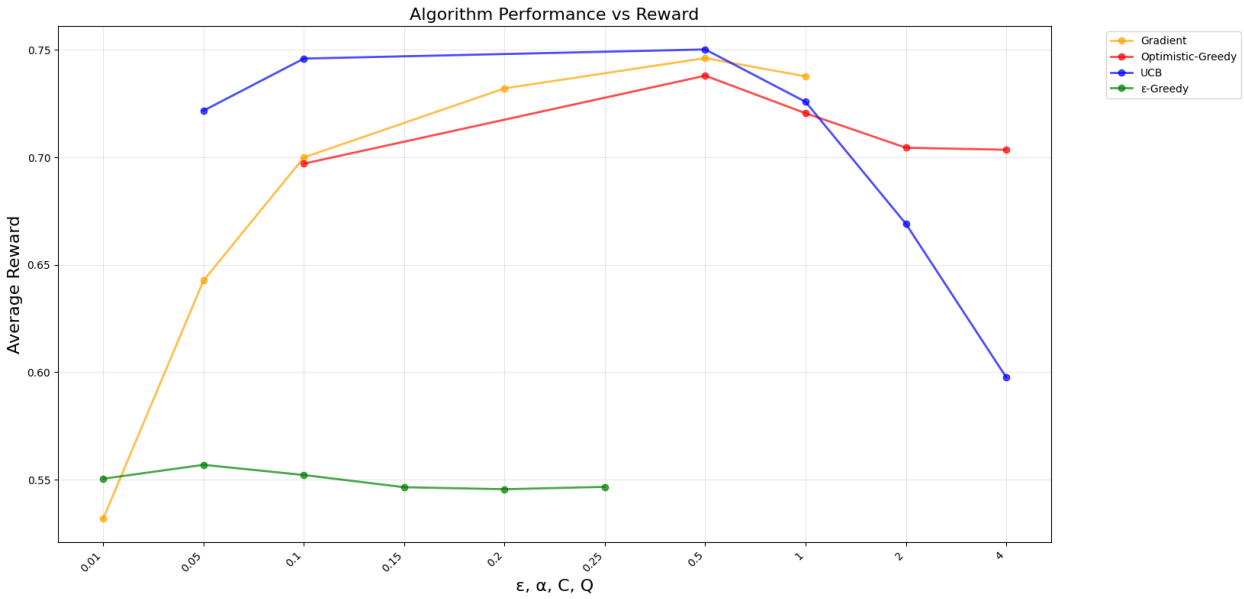
Realizacja eksperymentów polegała na uruchomieniu istniejącego już szkieletu kodu i badaniu, jak będą kumulowały się w czasie nagrody otrzymane przez poszczególne algorytmy.



Obraz 1. Porównanie nagród zgromadzonych w czasie przez różne algorytmy

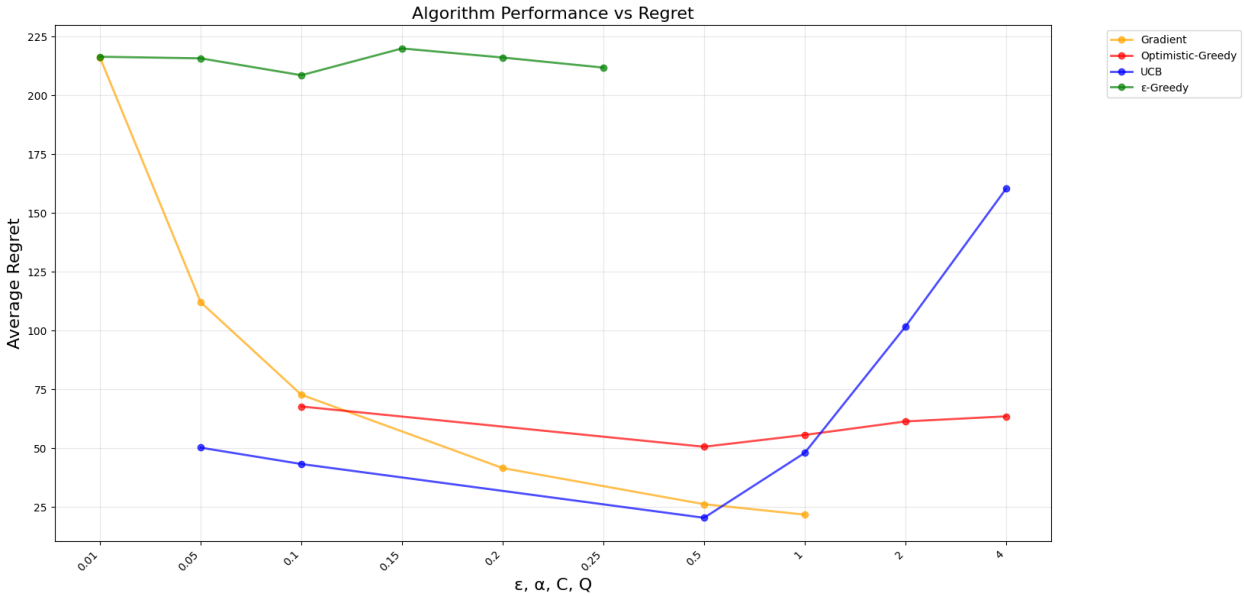
4. Studium parametryczne

1. Strategie vs Nagroda



Obraz 2. Studium parametryczne: Strategie vs Nagroda

2. Strategie vs Regret

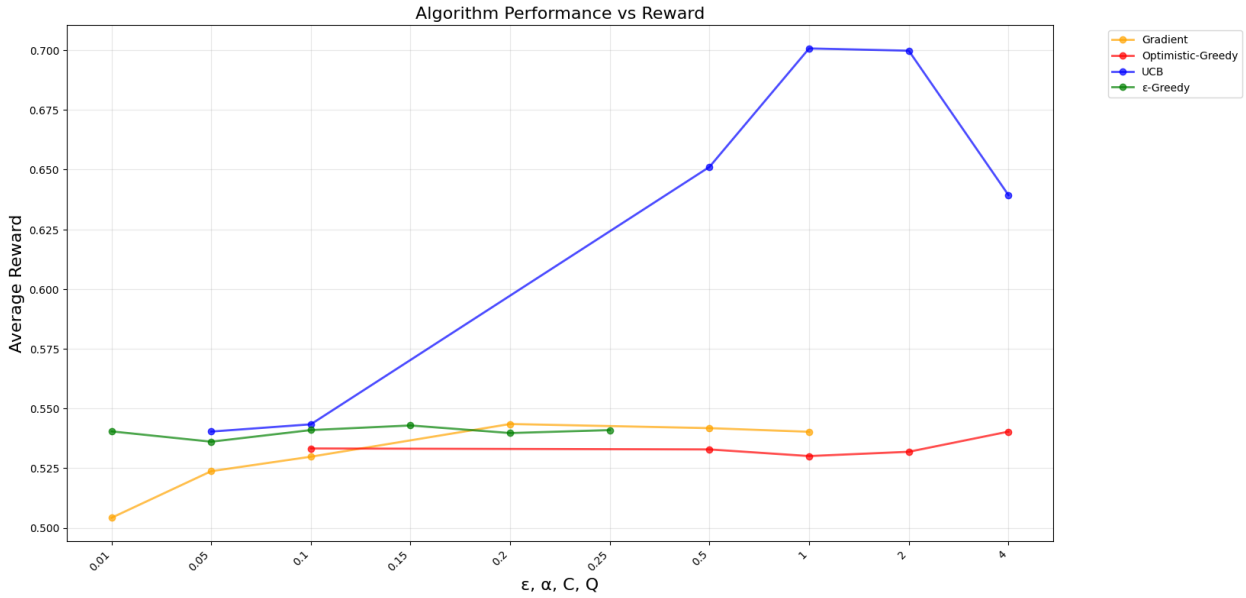


Obraz 3. Studium parametryczne: Strategie vs Regret

5. Bandyci niestacjonarni

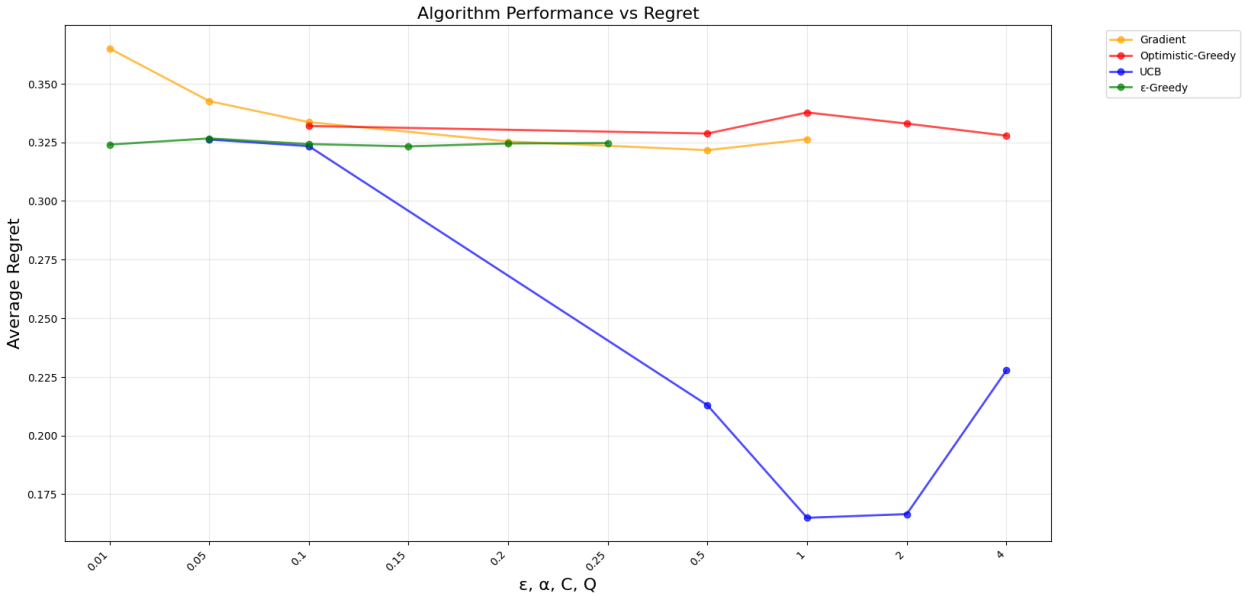
Dla wersji niestacjonarnej, w każdym kroku wartości nagród zostały zaktualizowane przy pomocy funkcji np.random.normal(0, 0.1, 1) o losową wartość z rozkładu normalnego o odchyleniu standardowym 0.1. Jeżeli po aktualizacji wartość nie mieściły się w przedziale [0,1], zostaje im przypisana wartość krańcowa przedziału przy użyciu np.clip(a_min=0, a_max=1).

1. Strategie vs Nagroda



Obraz 4. Niestacjonarne studium parametryczne: Strategie vs Nagroda

2. Strategie vs Regret



Obraz 5. Niestacjonarne studium parametryczne: Strategie vs Regret

6. Wnioski

Zarówno w wersji stacjonarnej jak i niestacjonarnej algorytmem, który uzyskiwał najlepsze wyniki była strategia Upper Confidence Bound. Najlepszymi wartościami parametru **c** tego algorytmu były wartości 0.1 i 0.5 dla wersji stacjonarnej i 1 oraz 2 dla testu niestacjonarnego. Drugim najlepiej radzącym sobie algorytmem był Gradient, dla którego 0.5 oraz 1, okazały się najlepszymi wartościami parametru **learning rate**. Należy również zaznaczyć, że w teście niestacjonarnym wydajność i przewaga algorytmu gradientowego nad strategiami zachłannymi drastycznie spada. Najgorzej radzącymi sobie algorytmami były strategie zachłanne: e-greedy oraz optimistic-greedy. Co ciekawe, algorytm optimistic-greedy radził sobie całkiem nieźle w teście stacjonarnym, natomiast okazał się najgorszy w studium niestacjonarnym. Dla wersji niestacjonarnej, wyniki algorytmu e-greedy nieznacznie zwiększają się wraz z wzrostem parametru **e**, co ma sens, ponieważ częściej eksplorujemy więc możemy poznać nowy rozkład. Wszystkie algorytmy poradziły sobie gorzej w teście niestacjonarnym.