```
import numpy as np
import random
```

Simple Combinatorial Multi-Armed Bandit (CMAB)

1.Explanation:

- **Arms and rewards:** Each base arm has a random reward between 0 and 1 (simulated by `true_rewards`).
- **UCB:** The `UCB` class implements the Upper Confidence Bound algorithm, where each arm's selection is based on its estimated value and a confidence interval that decreases as more samples are collected.
- **Simulation loop:** In each round, a combinatorial set of `k` arms is selected randomly, and their rewards are accumulated. The UCB values for each arm are updated based on these rewards.
- **Reward calculation:** A Bernoulli distribution is used to simulate the reward feedback for each arm.

2.Output: The output will show the total rewards for each round and the final learned values of each arm after the simulation.

```
n_arms = 5 # Number of base arms
k = 3 # Number of arms in each combinatorial action
n_rounds = 1000 # Total number of rounds to simulate
# Random reward function (between 0 and 1) for each arm
true_rewards = np.random.rand(n_arms)

# Function to simulate the reward of a selected arm
def get_reward(arm_index):
    # Simulate Bernoulli feedback with some randomness
    return np.random.binomial(1, true_rewards[arm_index])

# Upper Confidence Bound (UCB) algorithm for CMAB
class UCB:
    def __init__(self, n_arms, k):
        self.n_arms = n_arms
        self.k = k
        self.counts = np.zeros(n_arms)  # Counts of selections for
each arm
        self.values = np.zeros(n_arms)  # Estimated values for each
arm
        self.total_rewards = np.zeros(n_arms)  # Total rewards
collected for each arm
        self.time = 0  # Total number of actions taken
    def select_action(self):
        # UCB formula: Choose arm with the highest UCB
        ucb_values = self.values + np.sqrt(2 * np.log(self.time + 1) /
(self.counts + 1))
        return np.argmax(ucb_values)
    def update(self, chosen_arm, reward):
        self.time += 1
        self.counts[chosen_arm] += 1
        self.total_rewards[chosen_arm] += reward
        self.values[chosen_arm] = self.total_rewards[chosen_arm] / 
self.counts[chosen_arm]

# Initialize the UCB algorithm for CMAB
ucb = UCB(n_arms, k)
# Simulate the CMAB process
for round in range(n_rounds):
    chosen_arms = random.sample(range(n_arms), k)  # Select k arms 
(combinatorial choice)
    total_reward = 0
    # Collect rewards for each arm in the selected combination
    for arm in chosen_arms:
        reward = get_reward(arm)
        ucb.update(arm, reward)
        total_reward += reward
    # Output results (optional)
    if round % 100 == 0:
        print(f"Round {round}: Total Reward {total_reward:.2f}")

# After all rounds, show the learned values
print("\nFinal estimated values of each arm:")
print(ucb.values)
```

```
Round 0: Total Reward 1.00
Round 100: Total Reward 2.00
Round 200: Total Reward 2.00
Round 300: Total Reward 2.00
Round 400: Total Reward 0.00
Round 500: Total Reward 3.00
Round 600: Total Reward 3.00
Round 700: Total Reward 3.00
Round 800: Total Reward 3.00
Round 900: Total Reward 2.00

Final estimated values of each arm:
[0.33167496 0.48047538 0.82769726 0.73166667 0.49403748]
```

Off-CMAB

The **Off-CMAB** framework is an offline version of the Combinatorial Multi-Armed Bandit (CMAB), which works by utilizing pre-collected offline data (a dataset of actions and corresponding feedback) rather than exploring the environment in real-time. The **Combinatorial Lower Confidence Bound (CLCB)** algorithm is commonly used for Off-CMAB. It combines pessimistic estimations of rewards for base arms with a combinatorial solver. Below is the Python code for **Off-CMAB** using the **CLCB algorithm**, assuming we have a pre-collected dataset. The code leverages a given dataset of actions and feedback to select a combinatorial action that maximizes rewards.

```
# Simulated environment for generating rewards
def get_reward(arm_index, true_rewards):
    # Simulate Bernoulli feedback (reward = 1 with true_rewards
probability)
    return np.random.binomial(1, true_rewards[arm_index])

# Offline dataset (simulated)
def generate_offline_dataset(n_samples, n_arms, k):
    dataset = []
    for _ in range(n_samples):
        chosen_arms = random.sample(range(n_arms), k)  # Randomly
select k arms for this sample
        rewards = [get_reward(arm, true_rewards) for arm in
chosen_arms]  # Get rewards for selected arms
        dataset.append((chosen_arms, rewards))
    return dataset

# Combinatorial Lower Confidence Bound (CLCB) for Off-CMAB
class OffCMAB:
    def __init__(self, n_arms, k, dataset):
        self.n_arms = n_arms
        self.k = k
        self.dataset = dataset
        self.counts = np.zeros(n_arms)  # How many times each arm has
been selected
        self.total_rewards = np.zeros(n_arms)  # Total rewards for
each arm
        self.values = np.zeros(n_arms)  # Estimated values (mean
rewards) for each arm
        self.time = 0  # Total number of actions processed
    def compute_lcb(self, arm_index):
        # Calculate the lower confidence bound (LCB) for the arm
        if self.counts[arm_index] == 0:
            return float('inf')  # If arm hasn't been selected, set a
high LCB
        lcb = self.values[arm_index] - np.sqrt(np.log(self.time + 1) /
(2 * self.counts[arm_index]))
        return lcb
    def select_action(self):
        # Select the arm with the lowest LCB
        lcb_values = [self.compute_lcb(arm) for arm in
range(self.n_arms)]
        return np.argmin(lcb_values)
    def update(self, chosen_arms, rewards):
        # Update the counts, total rewards, and estimated values of
the chosen arms
        for arm, reward in zip(chosen_arms, rewards):
            self.counts[arm] += 1
            self.total_rewards[arm] += reward
            self.values[arm] = self.total_rewards[arm] / 
self.counts[arm]
        self.time += 1  # Increment the time step
    def train(self):
        # Train the Off-CMAB using the offline dataset
        for chosen_arms, rewards in self.dataset:
            self.update(chosen_arms, rewards)

# Simulate the true rewards for each arm
n_arms = 5
true_rewards = np.random.rand(n_arms)  # Simulated true reward for 
each arm
# Simulate offline dataset (using 100 samples)
n_samples = 100
k = 3  # Number of arms selected in each combinatorial action
dataset = generate_offline_dataset(n_samples, n_arms, k)
# Initialize Off-CMAB algorithm
off_cmab = OffCMAB(n_arms, k, dataset)
# Train Off-CMAB on the offline dataset
off_cmab.train()
# Display the learned values of each arm
print("Final estimated values of each arm (LCB values):")
print(off_cmab.values)
# Select a combinatorial action based on the LCB values
chosen_action = random.sample(range(n_arms), k)  # For simplicity, 
select a random action from the arms
print(f"\nChosen arms for the final action: {chosen_action}")

Final estimated values of each arm (LCB values):
[0.11111111 0.94736842 0.921875   0.72307692 0.11764706]

Chosen arms for the final action: [3, 1, 2]
```

1. Explanation:

- **Offline Dataset Generation**: The `generate_offline_dataset()` function simulates an offline dataset with `n_samples` entries. For each sample, `k` arms are randomly chosen, and their rewards are generated based on `true_rewards`.

- **Off-CMAB** :

  - **Initialization**: The class stores the number of arms (`n_arms`), the number of arms selected per action (`k`), and the offline dataset.
  - **LCB Calculation**: The `compute_lcb()` method computes the lower confidence bound (LCB) for a given arm, based on the number of times it has been selected and its observed rewards.

  - **Action Selection**: The `select_action()` method selects the arm with the smallest LCB, which corresponds to the arm that has the least uncertainty in terms of expected reward.
  - **Update**: The `update()` method updates the arm statistics after each sample in the dataset is processed.
- **Training**: The `train()` method trains the algorithm using the offline dataset, iterating through each sample and updating the arm statistics based on the observed rewards.

- **Simulated Rewards**: The `true_rewards` array holds the true reward probability for each arm. The rewards are simulated using a Bernoulli distribution, where each arm's reward is a success with probability equal to its corresponding true reward.

- **Final Action Selection**: After training, the estimated values for each arm are printed, and a final action (combinatorial selection of `k` arms) is made based on these values.

1. Output: The output will display the learned values for each arm after training (based on the offline dataset) and the final selection of `k` arms based on these values.

2. Note: This is only a demo, for it has a simple reward model and small datasets.

Cascading bandits (CB)

Cascading Bandits is a problem where you have a sequence of items (such as a list of recommendations) and a user interacts with them in a sequential manner. In each round, the user examines the items in the order presented, and once they find a satisfactory item, they stop interacting with the remaining items. This feedback is referred to as cascading feedback. The goal of the bandit algorithm is to learn the best ordering of items (the "ranked list") to maximize user engagement. The following is model of `Cascading Bandits`:

1. **Sequential Decision Making**: The learner will recommend a list of items to a user.
2. **Cascading Feedback**: The feedback is observed based on the user stopping at the first satisfactory item.

Note: Like `CMAB`, this part we still use `UCB` algorithm.

```
n_items = 5 # Number of items (base arms)
k = 3 # Length of the ranked list (number of items to recommend)
n_rounds = 1000 # Number of rounds (iterations)
# Simulated true reward function (probability of satisfaction for each
item)
true_rewards = np.random.rand(n_items)

# Function to simulate the reward for each item (whether the user is
satisfied or not)
def get_reward(item_index):
    # Simulate a Bernoulli feedback where reward is 1 (satisfied) with
probability `true_rewards[item_index]`
```

```
    return np.random.binomial(1, true_rewards[item_index])

# Cascading Bandits using UCB
class CascadingBandits:
    def __init__(self, n_items, k):
        self.n_items = n_items
        self.k = k
        self.counts = np.zeros(n_items)  # Number of times each item
has been shown
        self.total_rewards = np.zeros(n_items)  # Total rewards for
each item
        self.values = np.zeros(n_items)  # Estimated values (mean
rewards) for each item
        self.time = 0  # Total number of actions taken
    def compute_lcb(self, item_index):
        # Lower Confidence Bound (LCB) for the item
        if self.counts[item_index] == 0:
            return float('inf')  # If the item has never been shown,
assign a high LCB
        lcb = self.values[item_index] - np.sqrt(np.log(self.time + 1)
/ (2 * self.counts[item_index]))
        return lcb
    def select_action(self):
        # Select the item with the lowest LCB (to reduce uncertainty)
        lcb_values = [self.compute_lcb(i) for i in
range(self.n_items)]
        return np.argmin(lcb_values)
    def update(self, shown_items, rewards):
        # Update the counts, total rewards, and estimated values for
the shown items
        for item, reward in zip(shown_items, rewards):
            self.counts[item] += 1
            self.total_rewards[item] += reward
            self.values[item] = self.total_rewards[item] /
self.counts[item]
        self.time += 1  # Increment the time step

    def train(self, rounds):
        # Train the Cascading Bandits model using UCB and cascading
feedback
        for round in range(rounds):
            # Randomly select k items to recommend (can be changed to
a more sophisticated selection)
            recommended_items = random.sample(range(self.n_items),
self.k)
            total_reward = 0
            # Simulate cascading feedback (user examines items in the
recommended order)
            for item in recommended_items:
                reward = get_reward(item)
                total_reward += reward
                if reward == 1:
                    break # Stop at first satisfactory item
            # Update the model based on the feedback
            self.update(recommended_items, [get_reward(item) for item
in recommended_items])
            if round % 100 == 0:
                print(f"Round {round}: Total Reward
{total_reward:.2f}")

# Initialize the Cascading Bandits model
cascading_bandits = CascadingBandits(n_items, k)
# Train the model with the given number of rounds
cascading_bandits.train(n_rounds)
# After training, display the learned values (UCB-based estimates of
item rewards)
print("\nFinal estimated values for each item (LCB values):")
print(cascading_bandits.values)
```

```
Round 0: Total Reward 1.00
Round 100: Total Reward 1.00
Round 200: Total Reward 1.00
Round 300: Total Reward 1.00
Round 400: Total Reward 1.00
Round 500: Total Reward 1.00
Round 600: Total Reward 1.00
Round 700: Total Reward 1.00
Round 800: Total Reward 1.00
Round 900: Total Reward 1.00

Final estimated values for each item (LCB values):
[0.57762938 0.22203673 0.57453936 0.90243902 0.44067797]
```

1. Explanation:

- **Simulated Environment**:
  - The `true_rewards` array holds the satisfaction probabilities for each item. Each item has a true probability of satisfying the user when recommended.
  - The `get_reward()` function simulates the feedback, where a reward of `1` means the user is satisfied, and `0` means they are not.
- **Cascading Bandits Algorithm**:
  - **UCB (Upper Confidence Bound)**: The `compute_lcb()` method computes the Lower Confidence Bound (LCB) for each item, which is a pessimistic estimate of the reward.
  - **Item Selection**: The `select_action()` method selects the item with the lowest LCB value (most uncertain).

  - **Update**: The `update()` method updates the statistics of the selected items based on the observed feedback.

- **Training**: The `train()` method simulates the cascading feedback process for `n_rounds` iterations. In each round, a set of `k` items is selected, and the user's satisfaction with these items is observed in order. If the user is satisfied with an item, they stop interacting, and the subsequent items are ignored.

- **Cascading Feedback**: In each round, the model selects `k` items. As the user interacts with the items in the order they are presented, the model updates based on the first item that satisfies the user (`reward == 1`).

1. Output:

- The model tracks the total reward for each round and updates its knowledge (estimated rewards) for each item. After all rounds, the learned values (LCBs) are displayed.

- The code will print the total rewards accumulated during each round and, at the end, the estimated values for each item based on the UCB algorithm.

1. Note:

- This is a simplified `Cascading Bandit` and assumes **independent rewards** for each item.
- In more complex settings, rewards could be dependent, then we can replace the random selection of items with greedy or Thompson Sampling selection strategy.