This part describes the offline data collection process for the combinatorial multi-armed bandit with probabilistically triggered arms (CMAB-T) framework. The dataset consists of feedback data, where each data sample includes a combinatorial action, a triggered set of base arms, and their corresponding outcomes. The dataset is pre-collected by an experimenter, with each sample generated independently from a distribution over feasible actions and feedback. The experimenter's data collection distribution is assumed to be unknown, and it is important to capture the frequency of observing each arm and its associated outcomes. The paper also introduces the data-triggering probability, which indicates the frequency of each arm being observed during the data collection process.

The performance of an offline learning algorithm in this framework is evaluated based on the **suboptimality gap**. This gap quantifies the difference in expected reward between the optimal action and the action chosen by the algorithm, under the assumption that the algorithm has access to an approximation oracle. The oracle helps approximate the optimal solution by providing a near-optimal action. The authors define the **α-approximate suboptimality gap**, which is used to measure the performance of the offline learning algorithm. The objective is to minimize this gap, ensuring that the selected action performs close to the optimal one with high probability.

The section further introduces the concept of **data coverage conditions**, which are used to assess the quality of the offline dataset. These conditions determine the required data coverage for accurately estimating the reward of the optimal action. Two conditions are proposed: the **infinity-norm** and the **1-norm** triggering probability modulated (TPM) data coverage conditions. These conditions help ensure that the dataset provides sufficient information to estimate the expected reward of the optimal action with a small suboptimality gap. The quality of the dataset plays a critical role in the efficiency of offline learning, as the better the coverage, the fewer samples are needed to achieve a near-optimal solution.

Related Formulas:

- Suboptimality Gap:

$$\mathrm{SubOpt}(S\hat{\circ};\alpha,I):=\alpha\cdot r(S^{i};\mu)-r(S\hat{\circ};\mu)$$

  where $r(S^{i};\mu)$ is the expected reward of the optimal action S^{i}, and $r(S\hat{\circ};\mu)$ is the reward of the chosen action $S\hat{\circ}$.

- Data Triggering Probability:

$$p^{i}_{Darm,DS}=E_{S\sim DS,X\sim Darm,\tau\sim D_{trig}(S,X)}[I(i\in\tau)]$$

  This represents the frequency of observing base arm i when selecting action S.

Pseudocode for Offline Learning Algorithm:

```
# Input: Dataset D, α-approximation oracle ORACLE, failure probability δ
# Output: Chosen combinatorial action Ŝ

for arm i in [m]:
```

```
    # Calculate number of times arm i was observed
    Ni = sum([1 for t in range(n) if i in τt])
    # Calculate empirical mean for arm i
    mu_hat_i = sum([Xt,i for t in range(n) if i in τt]) / Ni
    # Compute lower confidence bound (LCB) for arm i
    LCB(μi) = mu_hat_i - sqrt(log(4mn/δ) / (2Ni))

# Use oracle to find the best combinatorial action based on LCBs
S_hat = ORACLE(LCB(μ1), ..., LCB(μm))
return S_hat

# Create a sample dataset
# Define a simple oracle that selects the action with the maximum LCB.
import numpy as np

# Function to simulate a dataset for arms, where each arm has n
outcomes
def generate_dataset(m, n):
    """Generate a dataset for m arms, each having n outcomes drawn
from a uniform distribution [0, 1]."""
    return [np.random.uniform(0, 1, n) for _ in range(m)]

# Define the oracle function that selects the action with the maximum
LCB
def simple_oracle(LCBs):
    """Oracle that selects the action (arm) with the maximum LCB."""
    return np.argmax(LCBs)

# Compute lower confidence bounds (LCBs)
def compute_LCB(data, delta, n):
    """Compute the LCBs for each arm based on the dataset."""
    m = len(data)  # Number of arms
    LCB = []
    for arm_data in data:
        Ni = len(arm_data)
        mean = np.mean(arm_data)
        lcb = mean - np.sqrt(np.log(4 * m * n / delta) / (2 * Ni))
        LCB.append(lcb)
    return LCB

# The CLCB algorithm implementation
def CLCB_algorithm(dataset, oracle, delta, n):
    """The CLCB algorithm to select the best action based on LCBs."""
    # Compute LCBs for all arms
    LCBs = compute_LCB(dataset, delta, n)

    # Use the oracle to select the best action based on LCBs
    S_hat = oracle(LCBs)

    return S_hat
```

```
# Parameters
m = 5  # Number of arms
n = 100  # Number of rounds (samples)
delta = 0.1  # Failure probability

# Generate a sample dataset of m arms and n samples for each arm
dataset = generate_dataset(m, n)

# Run the CLCB algorithm to select the best action
best_action = CLCB_algorithm(dataset, simple_oracle, delta, n)

print(f"The selected best action is arm {best_action + 1} (0-based
index).")
```

```
The selected best action is arm 2 (0-based index).
```

1. **Dataset**: We generate a dataset with `m` arms, and each arm has `n` samples (outcomes). For simplicity, each outcome is drawn randomly from a uniform distribution between 0 and 1.
2. **Oracle**: The `simple_oracle` function selects the action with the highest LCB from the list of LCBs computed for each arm.
3. **CLCB_algorithm**: This is the main function that calculates the LCBs for each arm and uses the oracle to select the best arm.
4. **Parameters**:
   - `m` (5): Number of arms.
   - `n` (100): Number of rounds or samples per arm.
   - `delta` (0.1): The failure probability used in the LCB calculation.

- The `CLCB_algorithm` will compute the LCB for each arm based on the dataset and select the arm with the highest LCB as the best action.
- The output will tell you which arm (indexed from 1) is selected as the best action based on the algorithm.