Programming Assignment 1

CS 747

Aditya Jadhav

2nd September, 2019

Implementation details of algorithms

Round Robin -

```python
def roundRobin(self,horizon):
    k = 0
    cumulativeReward = 0
    for turn in range(horizon):
        # Pull an arm and update the cumulative reward
        outcome = self.pullArm(k)
        cumulativeReward = cumulativeReward + outcome
        k = (k + 1) % self.variants

    maxReward = max(self.payouts) * horizon
    regret = maxReward - cumulativeReward
    return regret
```

The algorithm iterates over all the arms for the given time **horizon**. In each turn the current arm is pulled and the **cumulative reward** is updated based on the outcome of the arm.

ϵ Greedy -

```python
def epsilonGreedy(self,epsilon,horizon):
    cumulativeReward = 0

    empericalPayouts = [0] * self.variants
    rewards = [0] * self.variants
    pulls = [0] * self.variants

    # Estimating emperical payouts for n learning trials
    k = 0
    n = min(self.variants,horizon)
    for turn in range(n):
        # Pull kth arm and update the cumulative reward
        outcome = self.pullArm(k)
        cumulativeReward = cumulativeReward + outcome
        k = (k + 1) % self.variants

        # Update kth arms reward and pull count
        rewards[k] = rewards[k] + outcome
        pulls[k] = pulls[k] + 1
        empericalPayouts[k] = rewards[k]/pulls[k]


    # Explore / exploit with coresponding probabilites, for remaining turns
    for turn in range(n,horizon):
        # Choose whether to explore or exploit
        r = np.random.uniform(0,1)
        if (r < epsilon):
            # Explore : Choose and arm at random
            k = int(np.random.uniform(0,1) * (self.variants - 1))
        else:
            # Exploit : Choose the arm with maximum emperical payout
            k = empericalPayouts.index(max(empericalPayouts))

        # Pull kth arm and update the cumulative reward
        outcome = self.pullArm(k)
        cumulativeReward = cumulativeReward + outcome

        # Update kth arms reward and pull count
        rewards[k] = rewards[k] + outcome
        pulls[k] = pulls[k] + 1
        empericalPayouts[k] = rewards[k]/pulls[k]

    maxReward = max(self.payouts) * horizon
    regret = maxReward - cumulativeReward
    return regret
```

In this algorithm we first estimate the empirical payouts for **n learning trials**. Here **n** is an algorithm specific parameter and is chosen to be equal to the number of arms.

For the remaining turns, i.e **(horizon - n)** we try the exploration / exploitation approach. We choose to explore / exploit with probability ϵ and **(1 - ϵ)** respectively.

Once we have decided which arm to pull, we update the **cumulative reward** and also the **empirical payout** of pulled arm based on the outcome.

NOTE : The choice of n learning trials is programmer specific. Algorithm description from some sources may choose to omit these n learning trials. We have chosen implemented this here

UCB -

```python
def ucb(self,horizon):
    cumulativeReward = 0

    empericalPayouts = [0] * self.variants
    rewards = [0] * self.variants
    pulls = [0] * self.variants

    # Estimating emperical payouts for n learning trials
    k = 0
    n = min(self.variants,horizon)
    for turn in range(n):
        # Pull kth arm and update the cumulative reward
        outcome = self.pullArm(k)
        cumulativeReward = cumulativeReward + outcome
        k = (k + 1) % self.variants

        # Update kth arms reward and pull count
        rewards[k] = rewards[k] + outcome
        pulls[k] = pulls[k] + 1
        empericalPayouts[k] = rewards[k]/pulls[k]


    # Define upper confidence bound and use it for sampling
    ucbValues = [0] * self.variants
    for turn in range(n,horizon):
        for i in range(self.variants):
            term = math.sqrt(2 * math.log(turn) / pulls[i])
            ucbValues[i] = empericalPayouts[i] + term
        # Choose the arm with maximum ucb value and pull it
        k = ucbValues.index(max(ucbValues))
        outcome = self.pullArm(k)
        cumulativeReward = cumulativeReward + outcome

        # Update kth arms reward and pull count
        rewards[k] = rewards[k] + outcome
        pulls[k] = pulls[k] + 1
        empericalPayouts[k] = rewards[k]/pulls[k]

    maxReward = max(self.payouts) * horizon
    regret = maxReward - cumulativeReward
    return regret
```

In this algorithm we first estimate the empirical payouts for **n learning trials**. Here **n** is an algorithm specific parameter and is chosen to be equal to the number of arms.

For the remaining turns, i.e **(horizon - n)** we define upper confidence bound for all the arms as follows -

$$ucb_a^t = \hat{p}_a^t + \sqrt{\frac{2 \ln t}{u_a^t}}$$

We then pull the arm with maximum upper confidence bound. We then update the **cumulative reward** and the **empirical payout** of pulled arm based on the outcome of the arm.

KL UCB -

```python
def solve(self, pHat ,upperBound):
    # Define maxIter and precision
    maxIter = 25
    precision = 1e-6

    # Use binary search to find the optimum solution
    l = pHat
    r = 1
    for i in range(maxIter):
        m = (l + r) / 2
        kl = self.klDivergence(pHat,m)
        if (abs(kl - upperBound) < precision): break
        if (kl <= upperBound): l = m
        else: r = m
    return m

def generateKlUcbValues(self, empericalPayouts, turn, pulls):
    klUcbValues = [0] * self.variants
    for i in range(self.variants):
        # Get value using the below formula
        upperBound = (math.log(turn) + 3 * math.log(math.log(turn))) / pulls[i]
        klUcbValues[i] = self.solve(empericalPayouts[i],upperBound)
    return klUcbValues
```

Implementation is same as above. Only the formula for UCB changes -

$$ucb_a^t = max\,(q), \quad q \in [\,\hat{p}_a^t,\ 1\,] \quad \&$$

$$KL\,(\hat{p}_a^t, q) \leq \frac{\ln t + 3\ln(\ln t)}{u_a^t}$$

To solve this we use binary search. We fix algorithm specific parameters maxIter = 25 and precision = 10^{-6} for the solve function.

Thompson Sampling -

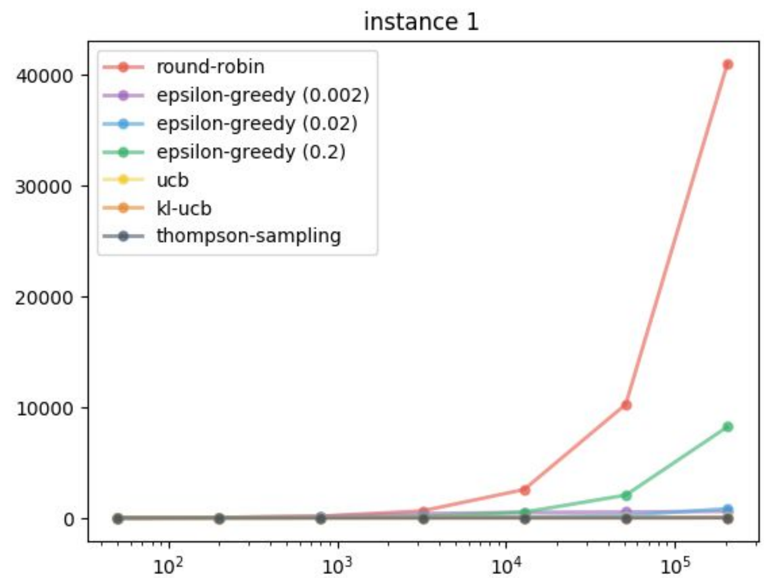
```python
def thompsonSampling(self,horizon):
    cumulativeReward = 0
    successes = [0] * self.variants
    faliures = [0] * self.variants
    for turn in range(horizon):
        # Calculate xa values by drawing from a beta distribution
        for i in range(self.variants):
            x = np.random.beta(successes[i]+1,faliures[i]+1)
        # Choose the arm with maximum x value
        k = x.index(max(x))
        outcome = self.pullArm(k)
        cumulativeReward = cumulativeReward + outcome

        # Update successes and faliures
        successes[k] = successes[k] + outcome
        faliures[k] = faliures[k] + int(not(outcome))

    maxReward = max(self.payouts) * horizon
    regret = maxReward - cumulativeReward
    return regret
```

In this algorithm, we keep track of number of **successes** and **failures** of each arm. In each turn we calculate $\mathbf{x_a^t}$ as follows -

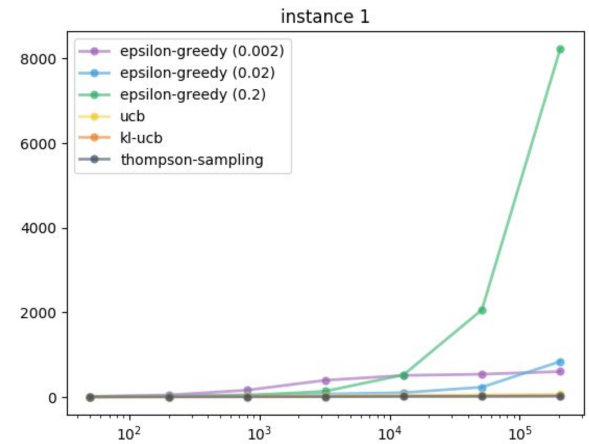
$$x_a^t \sim Beta\,(s_a^t + 1,\, f_a^t + 1)$$

We then select the arm with maximum $\mathbf{x_a^t}$ value. Once we have selected the arm, we pull it and update **cumulative reward**, **successes** and **failures** of that arm based on the outcome.

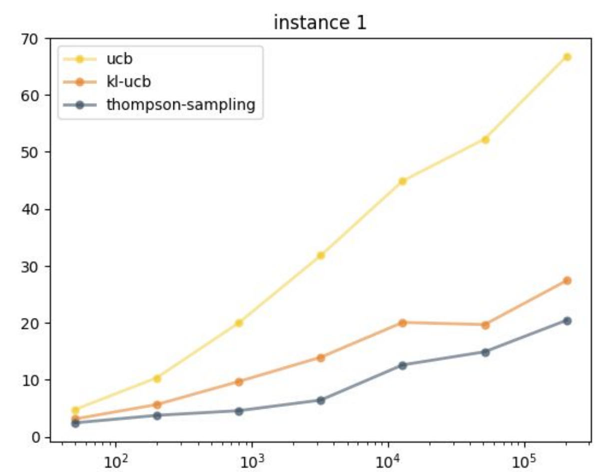
Performance analysis



(a)



(b)



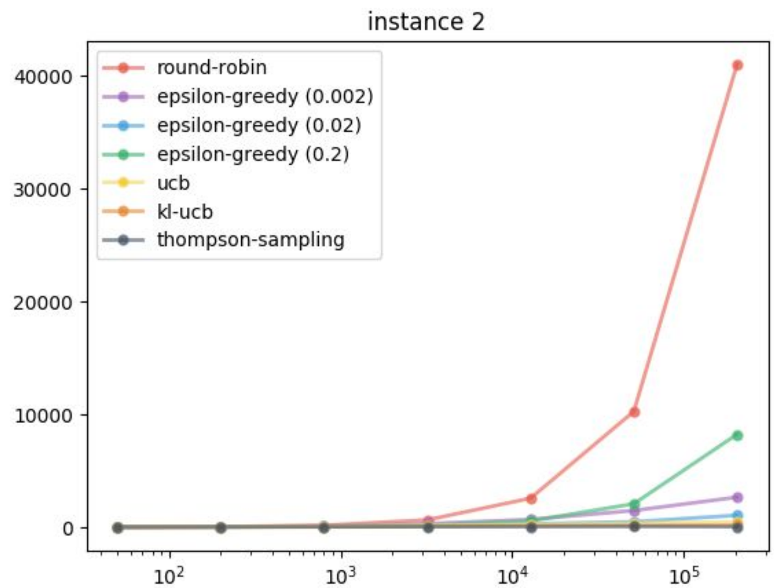
(c)

Details -

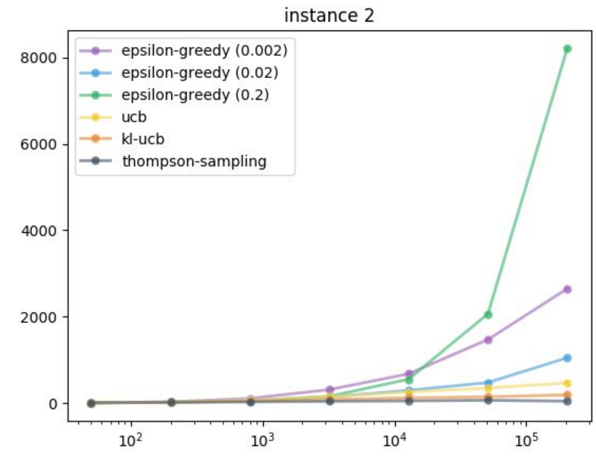
- Number of variants : **2**
- Payouts of variants : **[0.4, 0.8]**

Observations -

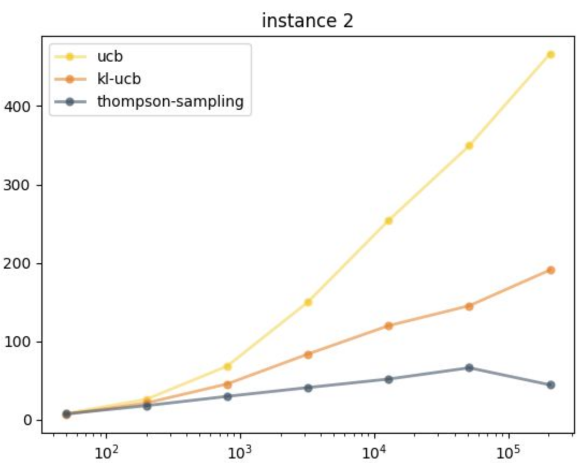
- From (a), round robin performs **worst out of all** the algorithms considered above.
- From (b), e-greedy 0.2 performs **worse out of the three** as it does a lot of unnecessary exploration.
- From (c), **not much conclusion** can be made about the performance of ucb, kl-ucb and thompson



(a)



(b)



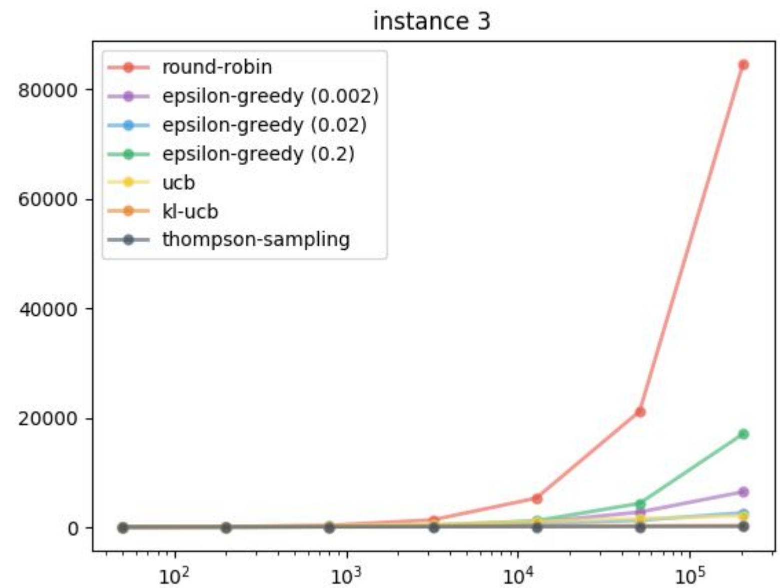
(c)

Details -

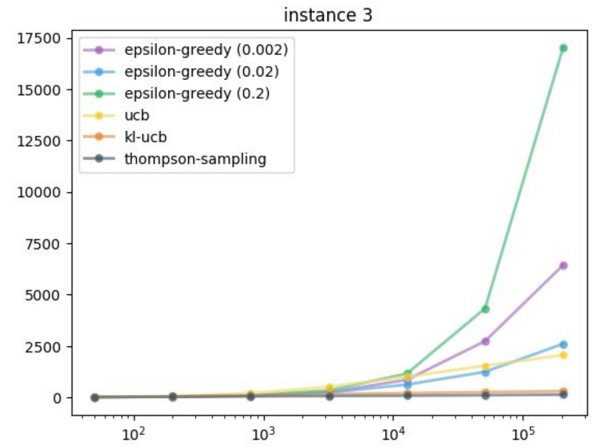
- Number of variants : **5**
- Payouts of variants : **[0.1, 0.2, 0.3, 0.4, 0.5]**

Observations -

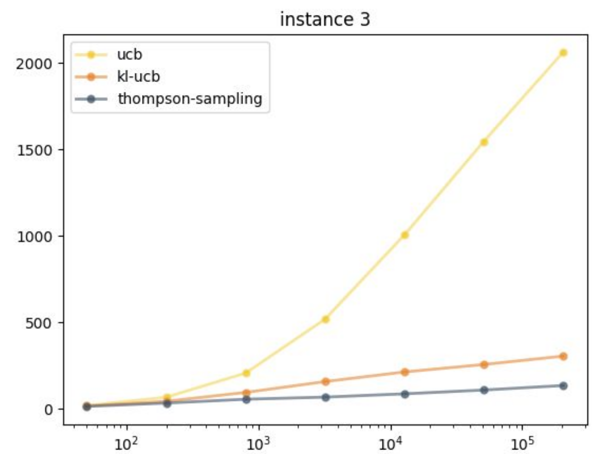
- From (a), round robin performs **worst out of all** the algorithms considered above.
- From (b), e-greedy 0.2 performs **worse out of the three** as it does a lot of unnecessary exploration.
- From (b), e-greedy 0.02 performs **best out of the three** as 5 variants need to be explored in this case.
- From (b), e-greedy 0.002 performs **worse than 0.02** as it does very less exploration.
- From (c), we can somewhat say that regret for **ucb is growing faster** than kl-ucb and thompson.



(a)



(b)



(c)

Details -

- Number of variants : **25**
- Payouts of variants : [**0.15, 0.23, 0.37, 0.44, 0.50, 0.32, 0.78, 0.21, 0.82, 0.56, 0.34, 0.56, 0.84, 0.76, 0.43, 0.65, 0.73, 0.92, 0.10, 0.89, 0.48. 0.96, 0.60, 0.54, 0.49**]

Observations -

- From (a), round robin performs **worst out of all** the algorithms considered above.
- From (b), e-greedy 0.2 performs **worse out of the three** as it does a lot of unnecessary exploration.
- From (b), e-greedy 0.02 performs **best out of the three** as 25 variants need to be explored in this case.
- From (b), e-greedy 0.002 performs **worse than 0.02** as it does very less exploration.
- From (c), regret for ucb seems to **grow exponentially** and hence should be **proportional to horizon**.
- From (c), regret for kl-ucb/thompson seems to **grow linearly** with horizon and hence should be **proportional to log(horizon)** as the scale on the x-axis is logarithmic .