CS747 - Assignment 1

Name - Utkarsh Ranjan
Roll No. - 200050147

Task 1

(1) UCB Algorithm

- For the UCB algorithm, ucb_a^t , u_a^t and $\hat{p}_a^t$ are first initialised with 0 for all Bandit instances.

```
# count initialized to 1 due count[i] being present as denominator in the
formula for ucb, in the long run it has diminising effect
self.counts = np.ones(num_arms)
self.ucb = np.zeros(num_arms)
self.values = np.zeros(num_arms)
```

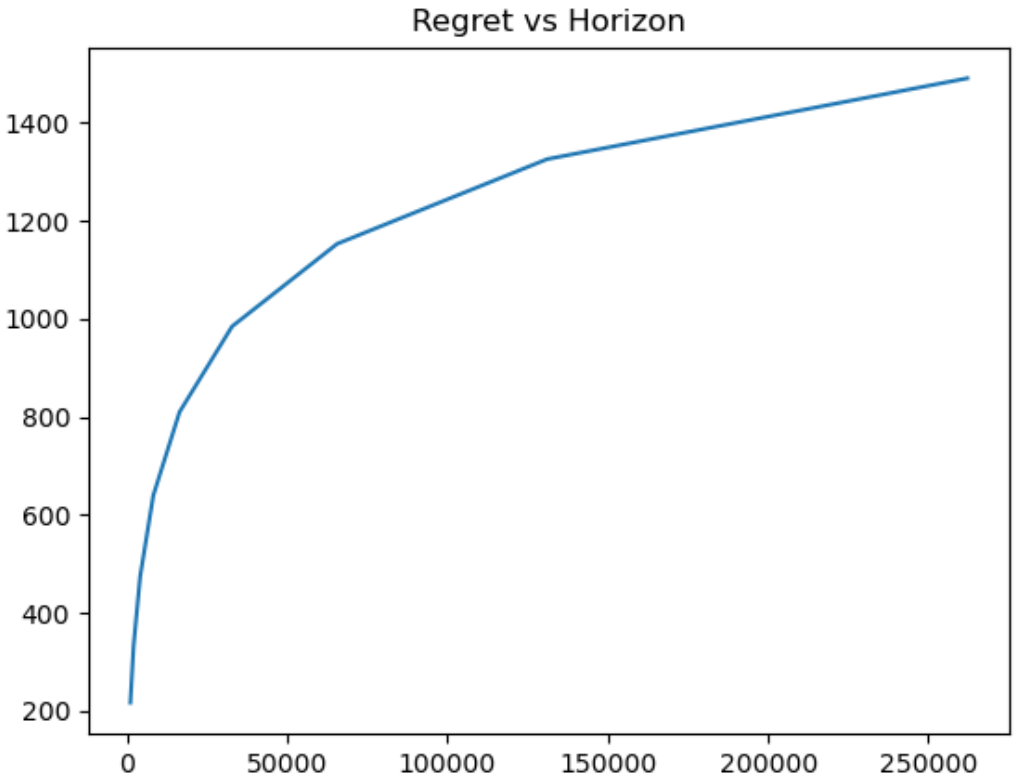
- For a particular pull, the bandit with the max value of ucb_a^t is chosen as the result

```
np.argmax(self.ucb)
```

- Based on the reward obtained for this Bandit - the value for this arm is updated. Also the ucb_a^t is updated for every bandit, due to increase in the time step 't'.

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

```
new_value = ((n - 1) / n) * value + (1 / n) * reward
self.ucb = self.values + np.sqrt(2 * np.log(self.horizon) *
np.reciprocal(self.counts))
```



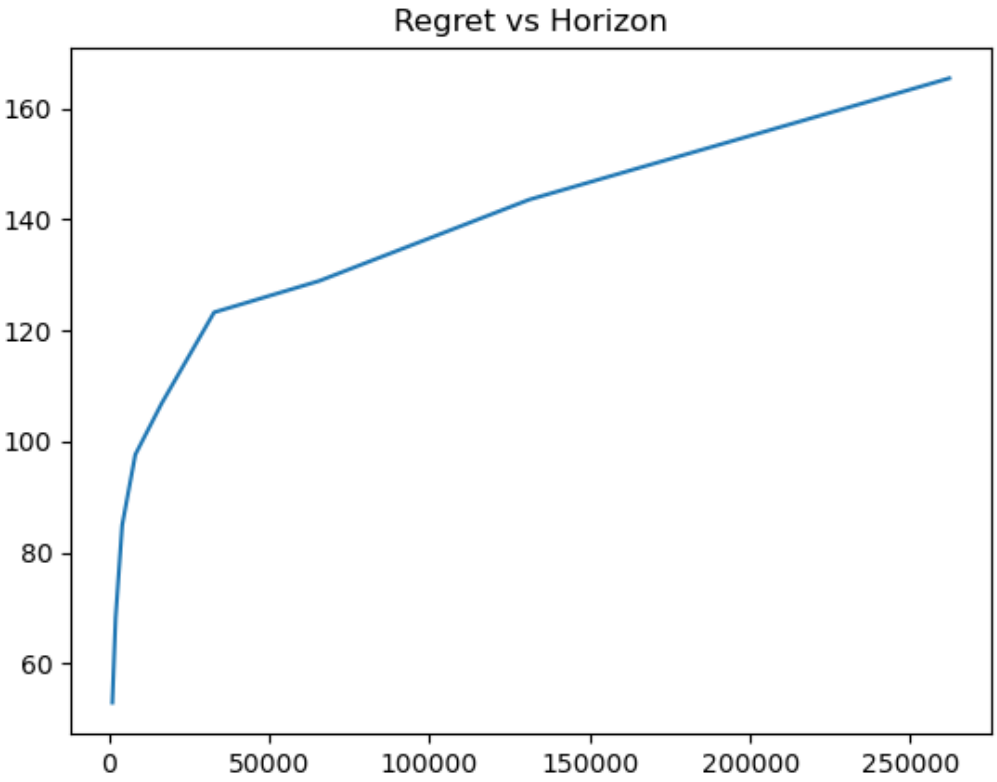
Due to the optimal action being identified quickly, and only trying the other actions when they have high uncertainty, the UCB method shows a much lower level of regret than seen in the Epsilon-Greedy method. The vast majority of the regret occurs during the initial priming round, where each bandit is being tried once to get its first estimate. Indeed, it has been shown that the expected cumulative regret of UCB is logarithmic in '*T*', the total number of time-steps.

(2) KL-UCB Algorithm

- The implementation for this algorithm is exactly similar to the ucb algorithm described above, except for a different definition of the upper confidence bound.

```
for i in range(self.num_arms):
    search_array = np.linspace(self.values[i], 1, 1000)
    assert self.counts[i] != 0
    self.kl_ucb[i] = binary_search( (math.log(self.horizon) + self.c *
math.log(math.log(self.horizon))) / self.counts[i], search_array,
self.values[i])
```

- The update of the kl-ucb requires search over a subset $[\hat{p}_a^t, 1]$ which was done with the implementation of Binary Search.



The plot demonstrate that KL-UCB is remarkably efficient and stable, including for short time horizons compared to its other competitors like UCB.

(3) Thompson Sampling

- For each bandit, the successes s_a^t and failures f_a^t array were initialised with 0.

```
self.success = np.zeros(num_arms)
self.failure = np.zeros(num_arms)
```

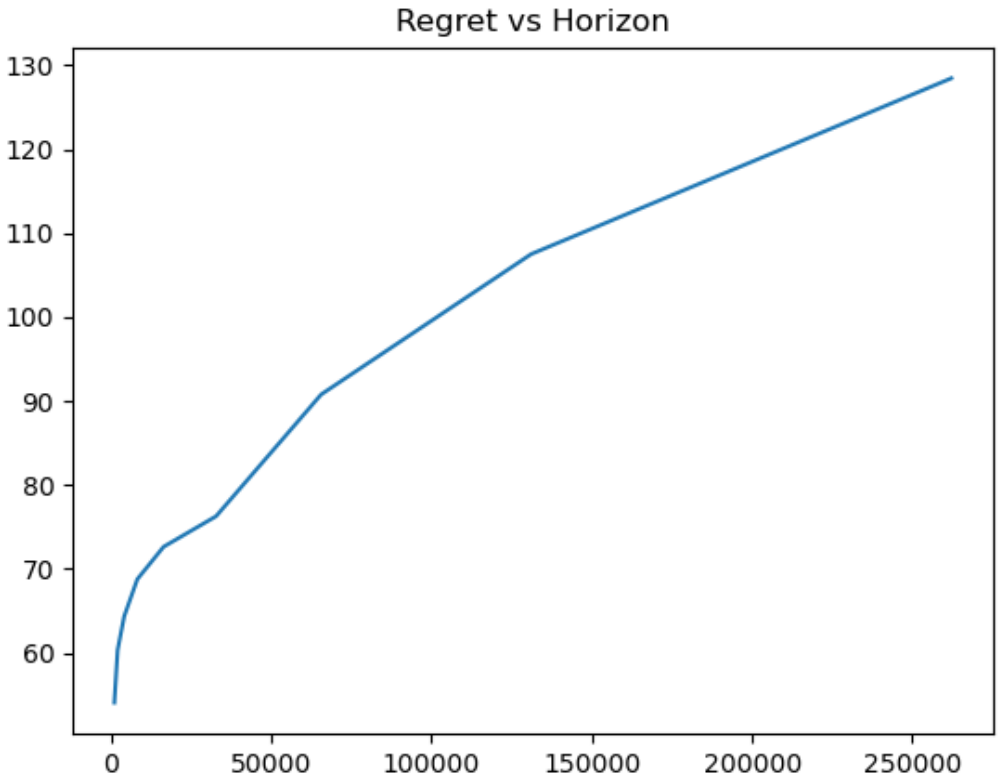
- The bandit instances were chosen based on the maximum sample obtained on sampling values for every bandit from beta distributions of the form :

$$Beta(s_a^t + 1, f_a^t + 1)$$

```
np.argmax(np.random.beta(self.success + 1, self.failure + 1))
```

- Based on the reward for the bandit pulled at a particular time step its success/failure counts were updated.

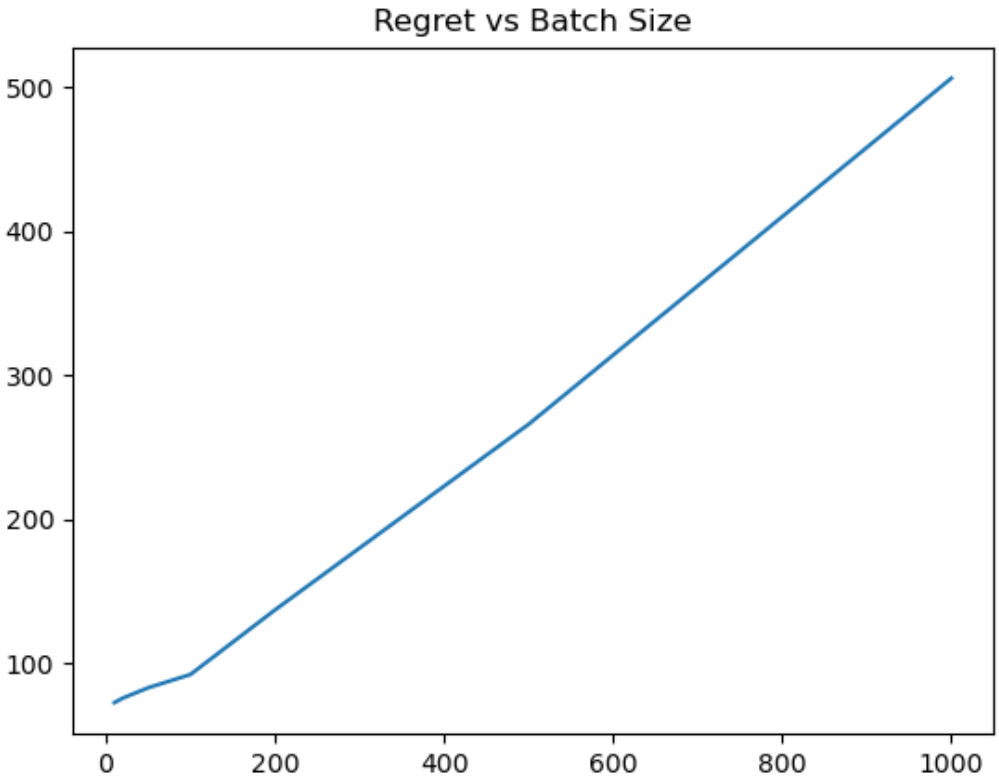
```
if reward == 1:
    self.success[arm_index] += 1
else:
    self.failure[arm_index] += 1
```



The plot of regret demonstrate that the performance of Thompson Sampling converges quickly to optimal, while that is far true for the greedy algortihm. This is because e-greedy is not judicious in how it selects paths to explore. The exploration is preformed uniformly in case of e-greedy. Thompson sampling on the other hand, orients exploration efforts towards informative rather than entirely random paths.

Task 2

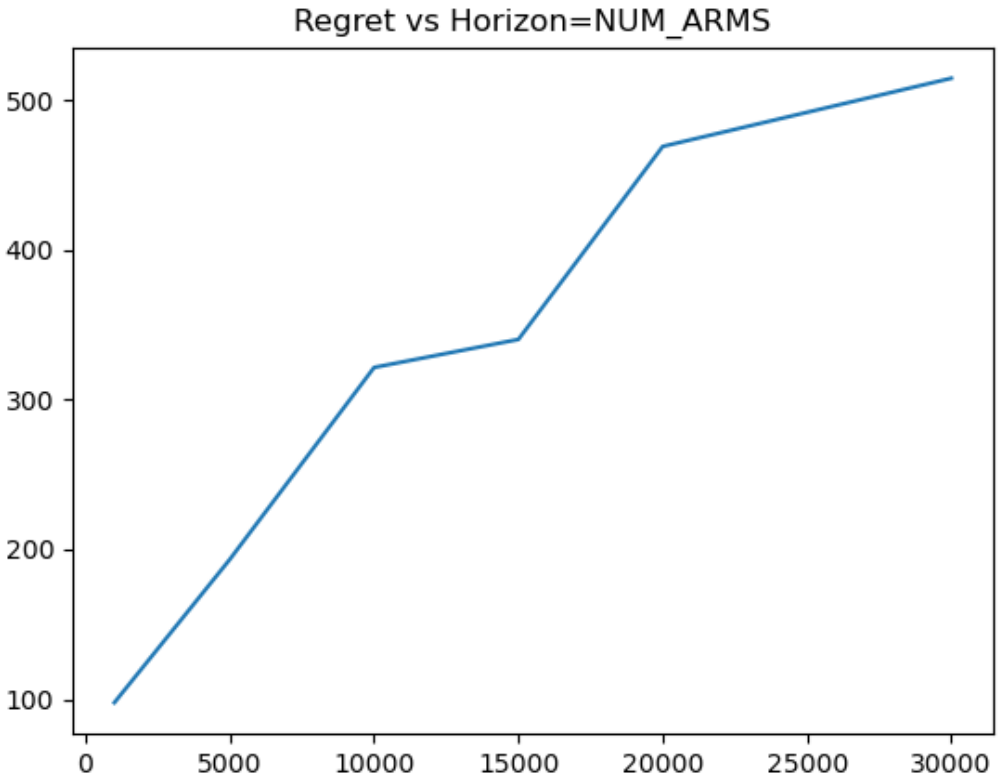
- A modification was made to the Thompson sampling algorithm to account for multi-bandit instances.
- For each bandit, the successes s_a^t and failures f_a^t array were initialised with 0 first.
- At every time step 't', a bandit arm was chosen 'batch_size' number of time, each time the selection was made using sampling random variables from beta distribution of individual bandits and taking argmax over all the bandit arms.
- Based on arm rewards obtained for the bandit pulled at a particular time step its success/failure counts were updated.



The graph depicts that the regret is increasing linearly with the batch size. When batch size = 1, it is simply the case of a the typical Thompson sampling algorithm, but as we increase the batch size, we start pulling many arms together without updating the success and failures for the previous pulls. This makes us devoid of the useful information we might have gained in the recent pulls. This leads to a general upward trend in the regret on increase in the batch size. If batch size will be equal / close to horizon, most of our pulls would be based on beta distribution which technically don't have any information about the Bernoulli probability of the arms, evidently leading to a higher regret.

Task 3

- This algorithm is also similar to Thompson sampling but a slight trick was used to achieve the results.
- Instead of pulling all the arms during the 'horizon' number of pulls, we instead considered only a fraction of all the arms (1/100 in this case) .
- So the experiment was performed as a regular Thompson Sampling, while considering only a fraction of total bandits for the experiment and ignoring the rest.



Because we are randomly using a fraction of the total number of bandits. The chances of choosing an arm with Bernoulli probability close to optimal arm reduces. As the horizon increases the number of bandits also increases since Horzon=NUM_ARMS, which leads to a poorer selection of the arm while picking a fixed fraction of existing arm. Thus performing Thompson sampling leads to an increase in regret with increase in the horizon.