IIT Bombay

Assignment 1
CS747 Intelligent & Learning Agents

Bandit Algorithms

Yash Sanjeev 180070068

8th Sep, 2021

Contents

1 Algorithms Implementation

All tasks have their own specific classes inheriting from a common base class `Algorithms` defined in the file `algorithms.py`. This class contains the general implementation for the 4 bandit algorithms we are concerned with in this assignment. The subsections are devoted to discussing this class, but let us first look at the `bandit.py` file which calls respective task functions depending on parameters passed.

```
import argparse

from task1 import BerSampling
from task2 import ScaleOptim
from task3 import GenSampling
from task4 import HighOptim

algo = ["epsilon-greedy-t1", "ucb-t1", "kl-ucb-t1",
↪  "thompson-sampling-t1", "ucb-t2", "alg-t3", "alg-t4"]
parser = argparse.ArgumentParser(description="bandit instance
↪  arguments")

parser.add_argument(
  "--instance", type=str,
  metavar="in", default=None,
  help="path to the instance file"
)

parser.add_argument(
  "--algorithm", type=str,
  metavar="al", default=None,
  help=" | ".join(algo)
)

parser.add_argument(
  "--randomSeed", type=int,
  metavar="rs", default=0,
  help="seed for random functions"
)

parser.add_argument(
  "--epsilon", type=float,
```

```
  metavar="ep", default=0.02,
  help="parameter in [0,1] for eps-greedy algos"
)

parser.add_argument(
  "--scale", type=float,
  metavar="c", default=2,
  help="scale for Task 2"
)

parser.add_argument(
  "--threshold", type=float,
  metavar="th", default=0,
  help="parameter in [0,1] for Task 4"
)

parser.add_argument(
  "--horizon", type=int,
  metavar="hz", default=0,
  help="horizon for different algos"
)

args = parser.parse_args()
task = args.algorithm.split("-")[-1]

assert args.instance, "please provide a file path"
assert args.randomSeed >= 0, "please provide a non-negative random
↪  seed"
assert args.horizon > 0, "please provide a positive bandit horizon"

if task == "t1":
  BerSampling(
    args.instance,
    args.algorithm,
    args.randomSeed,
    args.epsilon,
    args.scale,
    args.threshold,
    args.horizon
```

```
  )()

if task == "t2":
  ScaleOptim(
    args.instance,
    args.algorithm,
    args.randomSeed,
    args.epsilon,
    args.scale,
    args.threshold,
    args.horizon
  )()

if task == "t3":
  GenSampling(
    args.instance,
    args.algorithm,
    args.randomSeed,
    args.epsilon,
    args.scale,
    args.threshold,
    args.horizon
  )()

if task == "t4":
  HighOptim(
    args.instance,
    args.algorithm,
    args.randomSeed,
    args.epsilon,
    args.scale,
    args.threshold,
    args.horizon
  )()
```

The code simply calls the respective function depending on which task is being performed (present as the last character of the algorithm passed, for example, `ucb-t1` or `alg-t4`).

1.1 Attributes

```
class Algorithms:
  '''
    List of attributes needed
    - ins             file instance
    - al              algorithm to be used
    - rs              random seed
    - ep              epsilon
    - c               scale
    - th              threshold
    - hz              horizon
    - arms            number of arms
    - support         support of distribution
    - dist            distribution of the arms
    - cumdist         cumulative distribution
  '''
```

The only attributes that need some explanation is

1. `self.support` - This is a 2D `numpy` array. `self.support[i]` has the list of rewards corresponding to arm `i`.
2. `self.dist` - This is a 2D `numpy` array. `self.dist[i][j]` gives the probability of arm `i` producing reward `self.support[i][j]`.
3. `self.cumdist` - This is a 2D `numpy` array. `self.cumdist[i]` contains the cumulative distribution of arm `i` corresponding to the support. Thus, it is just the running sum of `self.dist` along axis 1.

1.2 Helper Functions

Armed with this information, we first take a look at the following helper functions (meant to be used form inside the class, not called by the user).

```
class Algorithms:
  def _sample(self, i):
    assert 0 <= i < self.arms, "arm number out of range"
    p               = np.random.random()
    rew             = self.support[np.searchsorted(self.cumdist[i],
    ↪  p)]
```

```
    self.rew      += rew
    total          = self.avg[i] * self.count[i] + rew
    self.count[i] += 1
    self.avg[i]    = total / self.count[i]

  def _search(self, i, t):
    assert 0 <= i < self.arms, "arm number out of range"

    dx = 1e-7
    f  = lambda x: min(max(x, dx), 1 - dx)
    kl = lambda p, q: f(p) * np.log(f(p) / f(q)) + f(1-p) *
     ↪  np.log(f(1-p) / f(1-q))

    avg, count = self.avg[i], self.count[i]
    start, end = avg, 1
    bound      = np.log(t) + self.c * np.log(np.log(t))

    while start <= end - dx:
      mid     = (start + end)/2
      if kl(avg, mid) > bound / count:
        end   = mid
      else:
        start = mid

    return (start + end)/2

  def _output(self):
    ins     = os.path.relpath(self.ins, os.path.dirname(__file__))

    if self.al == "alg-t4":
      reg   = np.max(np.sum((self.support > self.th) * self.dist,
       ↪  axis=1)) * self.hz - self.rew
      reg   = np.round(reg, 4)
      print(f"{ins}, {self.al}, {self.rs}, {self.ep}, {self.c},
       ↪  {self.th}, {self.hz}, {reg}, {self.rew}")

    else:
      reg   = np.max(np.sum(self.support * self.dist, axis=1)) *
       ↪  self.hz - self.rew
```

```
      reg   = np.round(reg, 4)
      print(f"{ins}, {self.al}, {self.rs}, {self.ep}, {self.c},
       ↪  {self.th}, {self.hz}, {reg}, 0")
```

Now let us elaborate on these helper functions.

1. `self._sample` - This function samples an arm `i`. First we randomly uniformly select a number `p` $\in [0, 1)$. Then, we map it to the index

$$\texttt{j} = \min\{\texttt{j} \leq \texttt{len(self.support)} | \texttt{self.cumdist[i][j]} \geq \texttt{p}\}$$

   . We define the reward to be `self.support[i][j]` which makes the probability of a reward happening exactly equal to its probability. We update the count and average reward of arm `i`. This function is used by all algorithms except Thompson Sampling, which uses success count rather than average rewards.

2. `self._search` - This function is used in KL-UCB to calculate

$$\min\{q \in [\hat{p}, 1) | D_{\text{KL}}(\hat{p}, q) \times \texttt{self.count[i]} > \log t + c \log\log t\}$$

   Since the KL Divergence in monotonic for $q > \hat{p}$, we use Binary Search in `self._search` and return the answer precise upto 7 decimal places.

3. `self._output` - This function produces the program output which is appended in the submitted output file.

1.3 Algorithms

With the helper functions out of the way, we finally look at the implementation of the bandit algorithms.

```
class Algorithms:
  def eps(self):
    self.rew       = 0
    self.avg       = np.zeros(self.arms)
    self.count     = np.zeros(self.arms)

    # sample each arm once at the start
    for i in range(self.arms):
      self._sample(i)
```

```
    for i in range(self.arms, self.hz):
      ep     = np.random.random()

      if ep  < self.ep:   # explore
        r    = np.random.random()
        arm  = int(r * self.arms)
      else:               # exploit
        arm  = np.argmax(self.avg)

      self._sample(arm)
    self._output()

    self._output()

  def ucb(self):
    self.rew       = 0
    self.avg       = np.zeros(self.arms)
    self.count     = np.zeros(self.arms)

    # sample each arm once at the start
    for i in range(self.arms):
      self._sample(i)

    for i in range(self.arms, self.hz):
      t   = i + 1    # round number
      ucb = self.avg + np.sqrt(self.c * np.log(t) / self.count)
      arm = np.argmax(ucb)
      self._sample(arm)

    self._output()

  def kl_ucb(self):
    self.rew       = 0
    self.avg       = np.zeros(self.arms)
    self.count     = np.zeros(self.arms)

    # sample each arm once at the start
    for i in range(self.arms):
      self._sample(i)
```

```
    for i in range(self.arms, self.hz):
      t      = i + 1    # round number
      kl_ucb = np.array([self._search(i, t) for i in range(self.arms)])
      arm    = np.argmax(kl_ucb)
      self._sample(arm)

    self._output()

  def thompson(self):
    self.rew     = 0
    self.success = np.ones(self.arms)
    self.failure = np.ones(self.arms)

    for i in range(self.hz):
      arm, val  = -1, -1

      for j in range(self.arms):
        a, b    = self.success[j], self.failure[j]
        sample  = np.random.beta(a, b)   # sample each arm
        if val  < sample:
          val   = sample
          arm   = j                      # choose best arm

      p         = np.random.random()
      rew       = self.support[np.searchsorted(self.cumdist[arm], p)]

      if rew > self.th:
        self.rew          += 1
        self.success[arm] += 1
      else:
        self.failure[arm] += 1
    self._output()
```

Time to elaborate on these implementations.

1. `self.eps` - is the implementation of the ϵ-greedy algorithm. We choose `ep` to be a random number between 0 and 1, so it is smaller than `self.ep` with probability `self.ep` corresponding to exploration. Thus, we choose a random arm when we explore and choose the arm with maximum `self.avg` when we exploit.

2. `self.ucb` - is the implementation of the UCB Algorithm. Initially, we sample each arm once. Thereafter we calculate the UCBs of all arms, and choose the arm corresponding to the maximum UCB. Then we call the helper function `self._sample(arm)` for that arm to sample it. This goes on until the horizon is reached.

3. `self.kl_ucb` - is the implementation of the KL-UCB algorithm. Here we calculate the KL-UCB using the helper function `self._search` for each arm and choosing the arm with the maximum KL-UCB. Everything else is exactly similar to the UCB algorithm so code is similar to `self.ucb`

4. `self.thompson` - is the implementation of Thompson Sampling. We maintain a tally of total successful and failed pulls for each arm in `self.success` and `self.failure` respectively. `self.success` is updated when the sampled reward is strictly greater than the threshold, otherwise `self.failure` is updated. This is exactly what we need in Task 4, however the same setting works as `self.th = 0` for Task 1, which means reward needs to be 1 for `self.success` to be updated.

2 Tasks

2.1 Task 1

The objective of Task 1 is to run each of the four algorithms for different instances over a range of random seeds, log the results in a file and plot the results when needed. The class which handles the code run for a particular algorithm with all given parameters is `BerSampling` situated in `task1.py`. The code for the class is pretty simple, which is expected since it inherits the `Algorithms` class.

2.1.1 Code

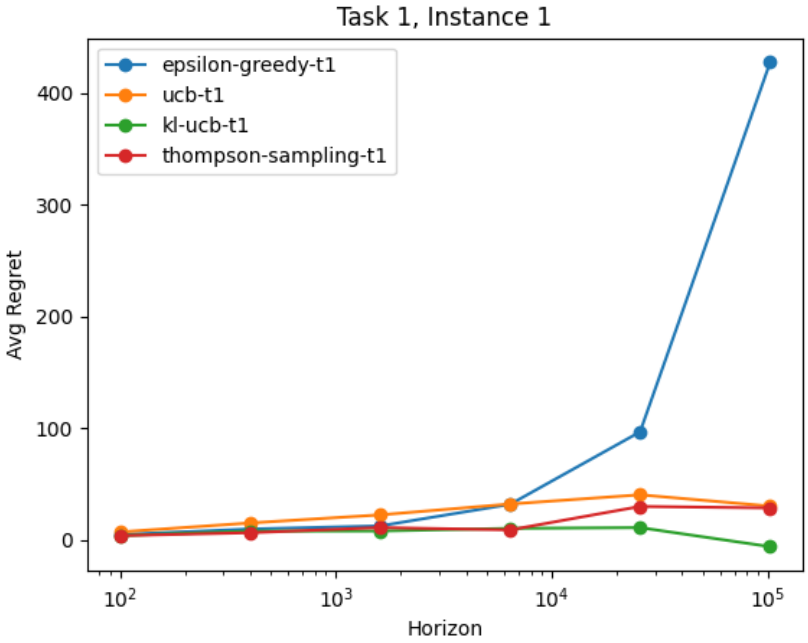
```
class BerSampling(Algorithms):
  def __init__(self, ins, al, rs, ep, c, th, hz):
    self.ins = ins  # file path
    self.al  = al   # algorithm
    self.rs  = rs   # random seed
    self.ep  = ep   # epsilon value
    self.c   = c    # scale (only for output)
    self.th  = th   # threshold (only for Thompson Sampling)
```

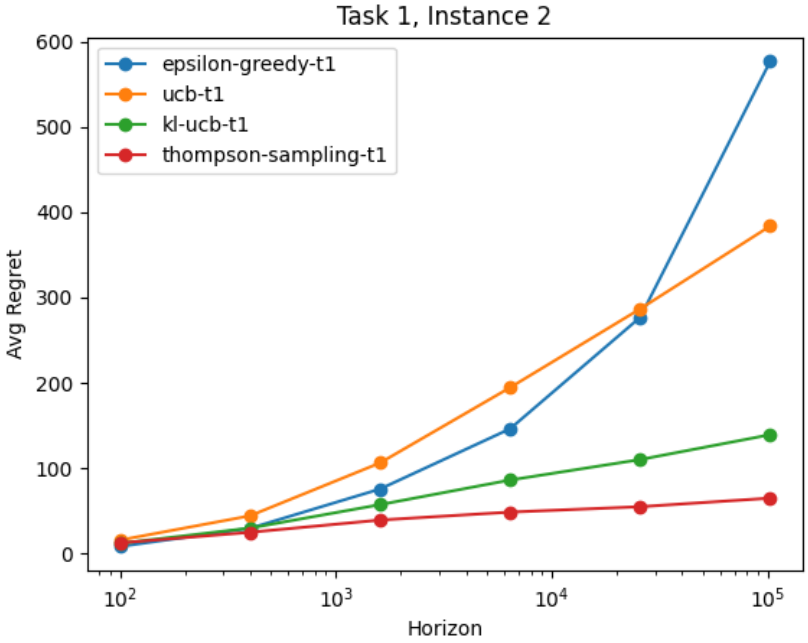
```
    self.hz  = hz   # horizon

    with open(self.ins) as f:
      lines          = f.readlines()
      self.arms      = len(lines)
      self.support   = np.array([0, 1])
      self.dist      = np.array([[1-float(l), float(l)] for l in lines])
      self.cumdist   = np.cumsum(self.dist, axis=1)

    # initialize the methods inherited from Algorithms class
    super().__init__()

  def __call__(self):
    funcmap = {
      "epsilon-greedy-t1": self.eps,
      "ucb-t1": self.ucb,
      "kl-ucb-t1": self.kl_ucb,
      "thompson-sampling-t1": self.thompson
    }

    np.random.seed(self.rs)
    assert self.al in funcmap, "incorrect algorithm specified in task 1"
    funcmap[self.al]()
```

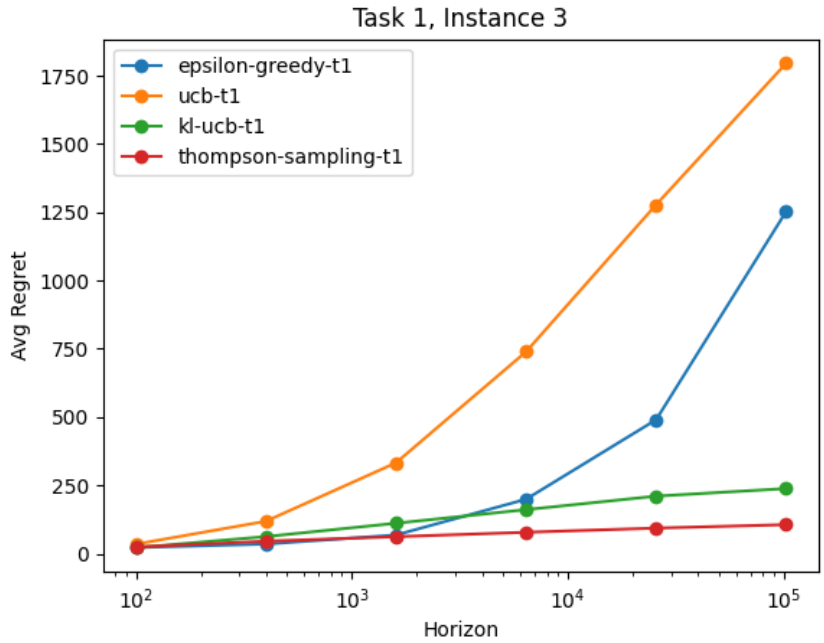
We initialise the various attributes in `self.__init__` and call the instance itself in `self.__call__` to run a specific algorithm from `class Algorithms` depending on `self.al`. The validity of the algorithms has been discussed in the section on implementation. A sample command to run task 1 is

```
python submission/bandit.py \
    --instance ../instances/instances-task1/i-1.txt \
    --algorithm epsilon-greedy-t1 \
    --epsilon 0.02 \
    --randomSeed 7 \
    --horizon 10000
```

2.1.2 Plots


Task 1, Instance 1
epsilon-greedy-t1
ucb-t1
kl-ucb-t1
thompson-sampling-t1
Avg Regret
Horizon
0
100
200
300
400
10^2
10^3
10^4
10^5


Task 1, Instance 2
epsilon-greedy-t1
ucb-t1
kl-ucb-t1
thompson-sampling-t1
Avg Regret
Horizon
0
100
200
300
400
500
600
10^2
10^3
10^4
10^5



2.1.3 Inference

Please observe that the X-axis has been plotted on the log scale, thus log trends appear linear and linear trends appear exponential.

1. ϵ-Greedy Algorithm - This algorithm has a linear average regret, which can be clearly seen by the exponential trend in each of the three instances we have plotted.

2. UCB - This algorithm has a logarithmic regret as horizon approaches infinity. However, we see the linear trend only in instance 1 which was a relatively easier instance as it had only two arms with a very large mean difference. I believe that the exponential trend (hence linear regret) observed in the other two instances is because the horizon was not large enough for the algorithm to converge to the optimal arm resulting in seemingly linear regret. Tuning the scale value could help in such cases, which is explored in detail in Task 2.

3. KL-UCB - This algorithm has a tighter bound than UCB and generally does better than it, even though the average regret is still logarithmic. The better results of this algorithm can be clearly seen in the plots where it achieves a linear trend (hence logarithmic regret) wrt horizon in all three instances.

4. Thompson Sampling - This algorithm does the best in the two relatively computationally harder instances 2 and 3, and also achieves linear trend in all three instances. This makes it the best algorithm for Bernoulli Bandit instances even though it has no parameters to be tuned. This motivated me to choose this algorithm as my weapon of choice for Task 4.

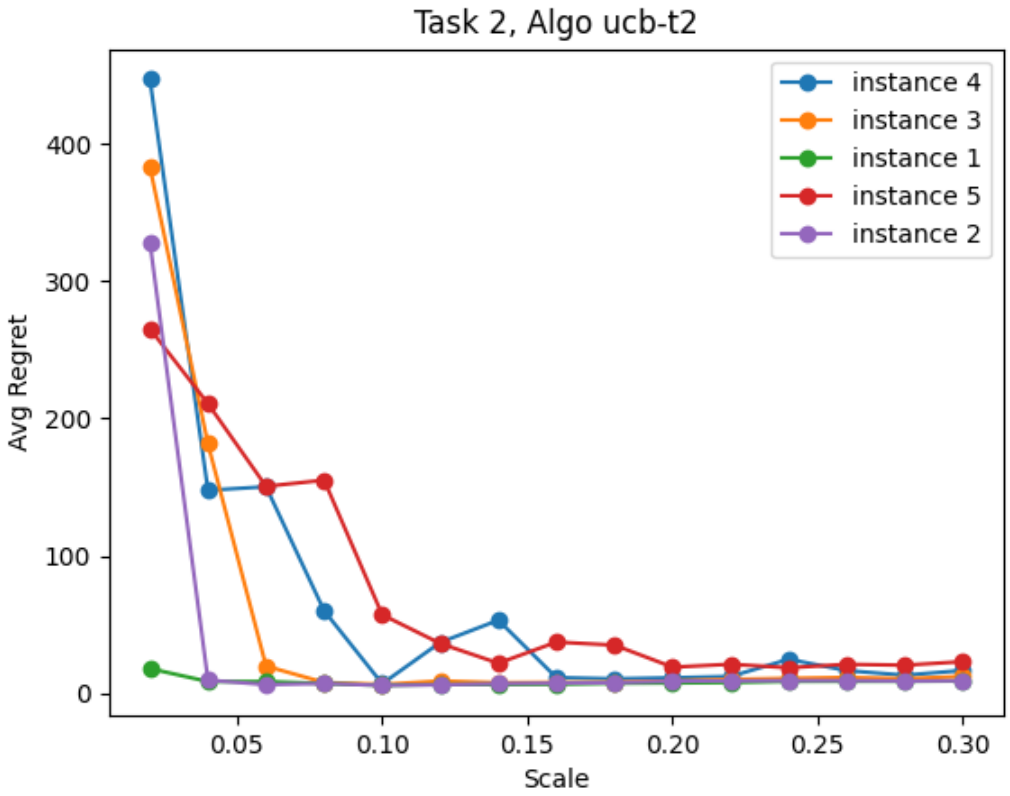
2.2 Task 2

This task comprised of tuning the scale value for the UCB Algorithm to achieve the least possible regret on certain fixed instances. The scale was ranged between 0.02 and 0.30, increased by an amount of 0.02 at each instance. The average regret for each scale corresponding to 50 random seeds was plotted against the scales themselves.

2.2.1 Code

```
class ScaleOptim(Algorithms):
  def __init__(self, ins, al, rs, ep, c, th, hz):
    self.ins = ins  # file path
    self.al  = al   # algorithm
    self.rs  = rs   # random seed
    self.ep  = ep   # epsilon value
    self.c   = c    # scale (only for output)
    self.th  = th   # threshold (only for output)
    self.hz  = hz   # horizon

    with open(self.ins) as f:
      lines          = f.readlines()
      self.arms      = len(lines)
      self.support   = np.array([0, 1])
      self.dist      = np.array([[1-float(l), float(l)] for l in
      ↪  lines])
      self.cumdist   = np.cumsum(self.dist, axis=1)

    # initialize the methods inherited from Algorithms class
    super().__init__()

  def __call__(self):
    np.random.seed(self.rs)
    self.ucb()
```

2.2.2 Plots

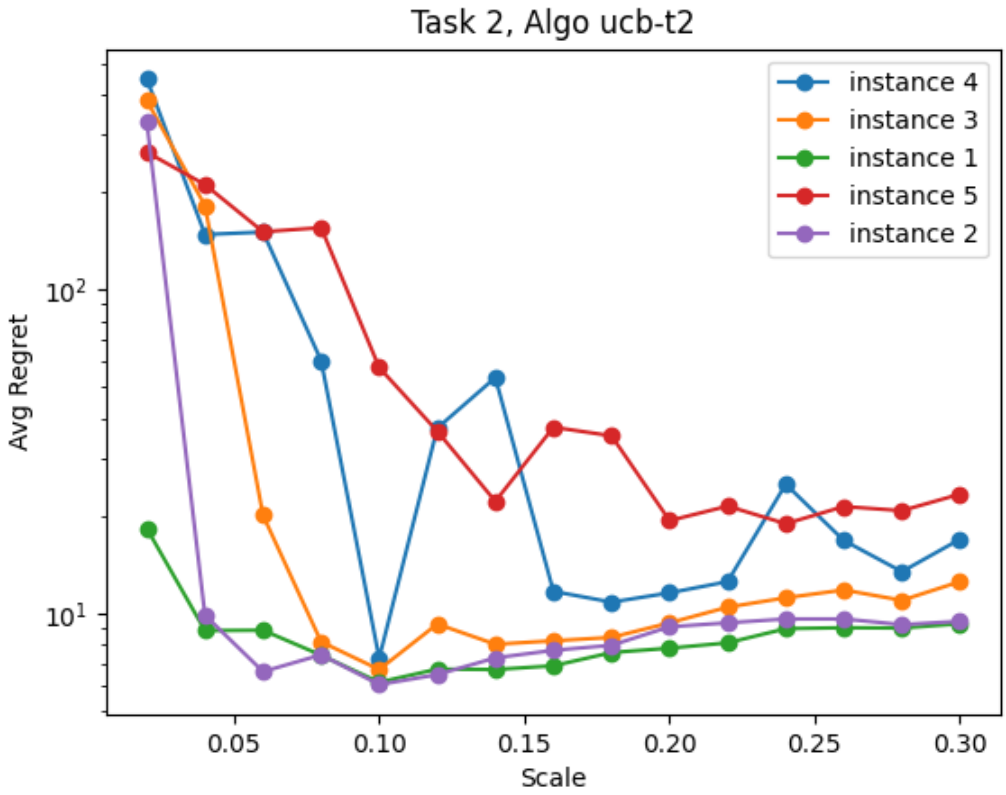


Instance	Optimal Scale
1	0.10
2	0.10
3	0.10
4	0.10
5	0.24

2.2.3 Inference

We plot the average regret on the log scale below as the large regrets for small scale values mask how the trend changes for larger scale values. From this plot, we see that for each instance, the regret first decreases with scale, reaches a min value and then increases. Remembering that larger scale corresponds to more exploration, we

see that this observation makes complete sense. For lower values of scale, we do not explore enough and hence end up sampling a suboptimal arm as the optimal arm resulting in very large regrets. For higher values of scale, we do not exploit enough, as larger confidence intevals force us to choose suboptimal arms more often. Thus, there is a sweet spot for the scale, which is unique to each instance, where the optimal amount of exploration and exploitation is done. It can be hpothesized that larger scale (hence more exploration) might be needed for more complex instances. This is corroborated by the largest optimal scale value for instance 5, which is the hardest though we also see that the same optimal scale value works for instances 1 to 4 even though they differ in complexity.



2.3 Task 3

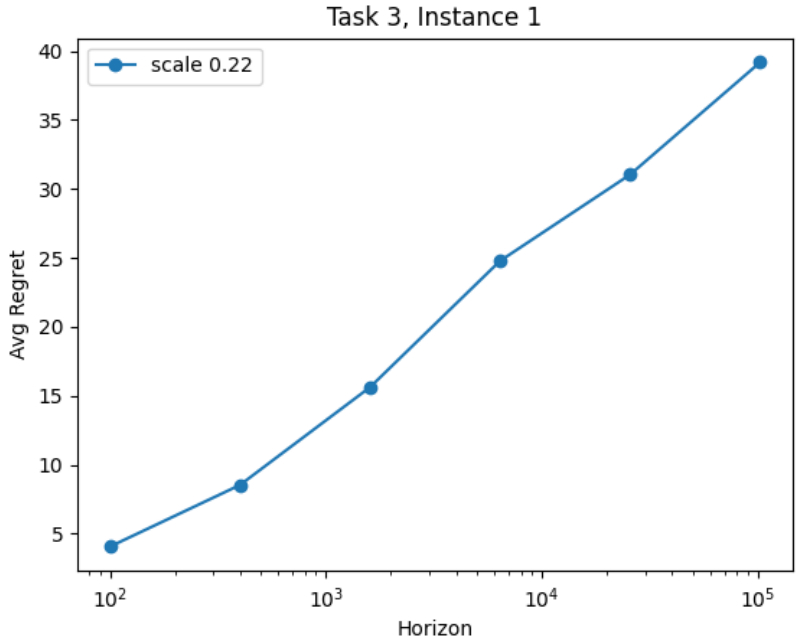
The objective of Task 3 is to minimise regret on finite support discrete distributions more general than the Bernoulli Distribution. I found it hard to alter Thompson

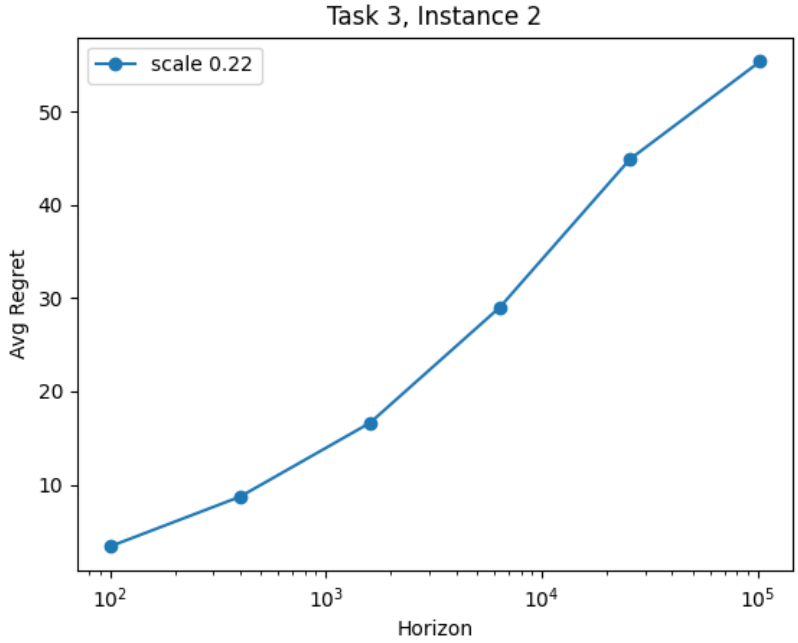
Sampling for the more general case, hence experimented with UCB and KL-UCB options. After some experiments, I concluded that UCB performs better (and is much faster) than KL-UCB, so I chose it as the algorithm for Task 3.

2.3.1 Code

```
class GenSampling(Algorithms):
  def __init__(self, ins, al, rs, ep, c, th, hz):
    self.ins = ins  # file path
    self.al  = al   # algorithm
    self.rs  = rs   # random seed
    self.ep  = ep   # epsilon value
    self.c   = c    # scale (only for output)
    self.th  = th   # threshold (only for output)
    self.hz  = hz   # horizon
    self.rew = 0    # cumulative reward

    with open(self.ins) as f:
      lines          = f.readlines()
      self.arms      = len(lines) - 1
      self.support   = np.array([float(l) for l in
      ↪  lines[0].strip().split()])
      self.dist      = np.array([[float(l) for l in
      ↪  line.strip().split()] for line in lines[1:]])
      self.cumdist   = np.cumsum(self.dist, axis=1)
      self.avg       = np.zeros(self.arms)
      self.count     = np.zeros(self.arms)

    # initialize the methods inherited from Algorithms class
    super().__init__()

  def __call__(self):
    assert self.al == "alg-t3", "incorrect algorithm specified in
    ↪  task 3"
    np.random.seed(self.rs)
    self.ucb()
```

2.3.2 Plots


Task 3, Instance 1
scale 0.22
Avg Regret
Horizon
40
35
30
25
20
15
10
5
10^2
10^3
10^4
10^5


Task 3, Instance 2
scale 0.22
Avg Regret
Horizon
50
40
30
20
10
10^2
10^3
10^4
10^5

I ranged the scale from 0.02 to 0.32, same as Task 2, and chose the optimal scale for each instance. For both the given instances, I found the optimal scale to be 0.22. Graphs corresponding to these scales is shown above.

2.3.3 Inference

Clearly, the same scale is not going to work on all types of instances, simpler ones will require lesser exploration compared to the more complex instances. Thus, this algorithm has a parameter that needs to be tuned for each instance. Not exactly desirable since we want our algorithm to work on a large variety of instances. For a general case, I believe KL-UCB with its default scale 3 will do much better than a fixed scale value for UCB.

2.4 Task 4

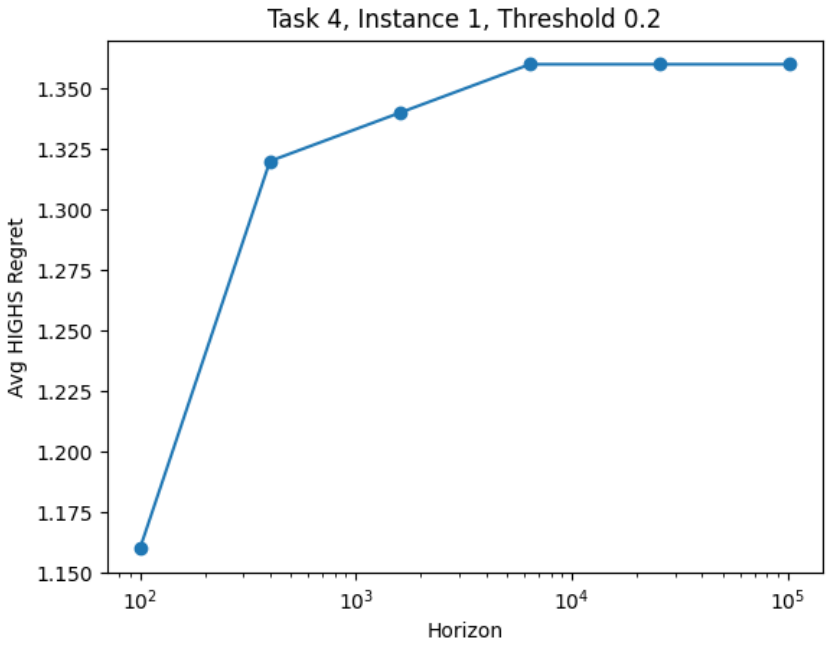
The objective here is to earn a reward greater than a given threshold the maximum number of times. Clearly the optimization metric is binary, since we either get a reward greater than threshold or we don't. Thus, the target random variable is binary, so all algorithms from Task 1 can be implemented in this case. Since Thompson Sampling performed the best in Task 1 without any tuning, I chose this algorithm for Task 4 as well.

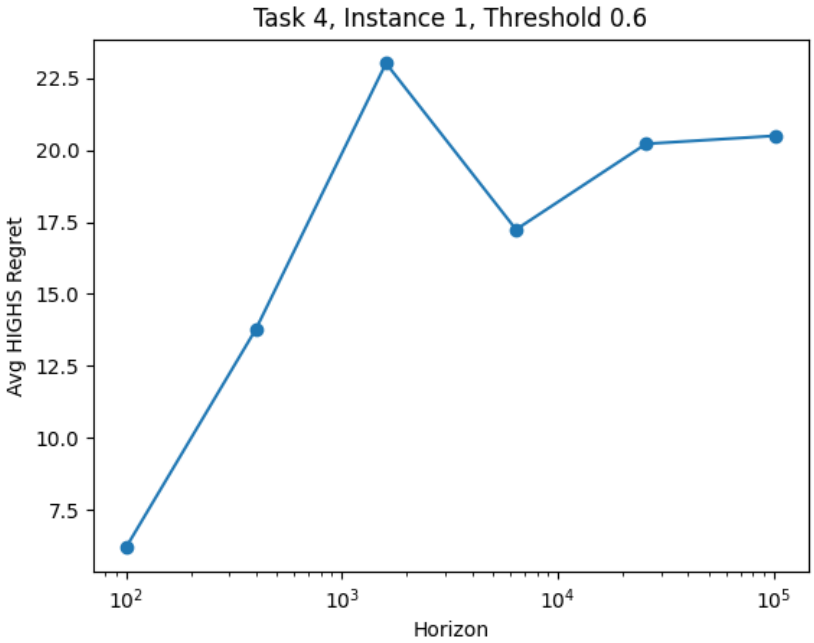
2.4.1 Code

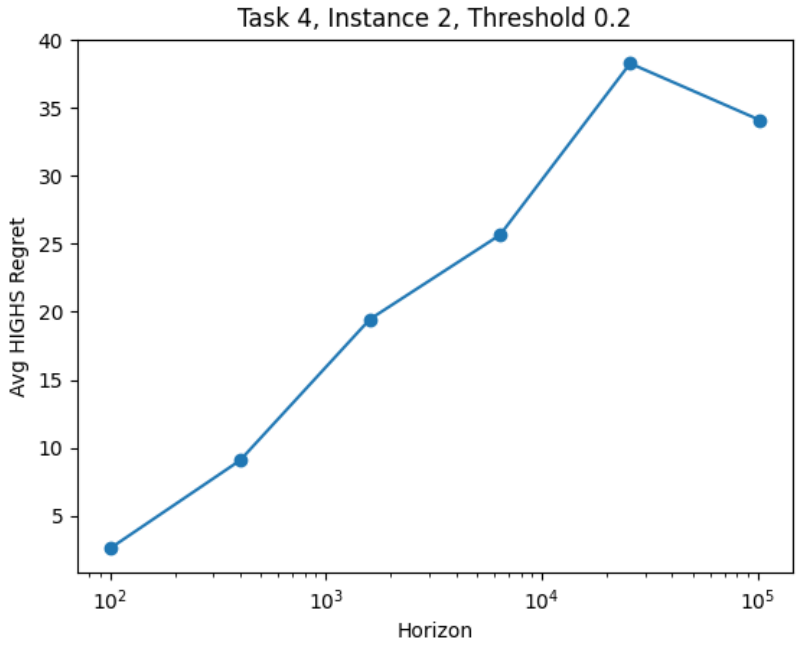
```
class HighOptim(Algorithms):
  def __init__(self, ins, al, rs, ep, c, th, hz):
    self.ins = ins  # file path
    self.al  = al   # algorithm
    self.rs  = rs   # random seed
    self.ep  = ep   # epsilon value
    self.c   = c    # scale (only for output)
    self.th  = th   # threshold (only for output)
    self.hz  = hz   # horizon

    with open(self.ins) as f:
      lines          = f.readlines()
      self.arms      = len(lines) - 1
      self.support   = np.array([float(l) for l in
       ↪  lines[0].strip().split()])
```

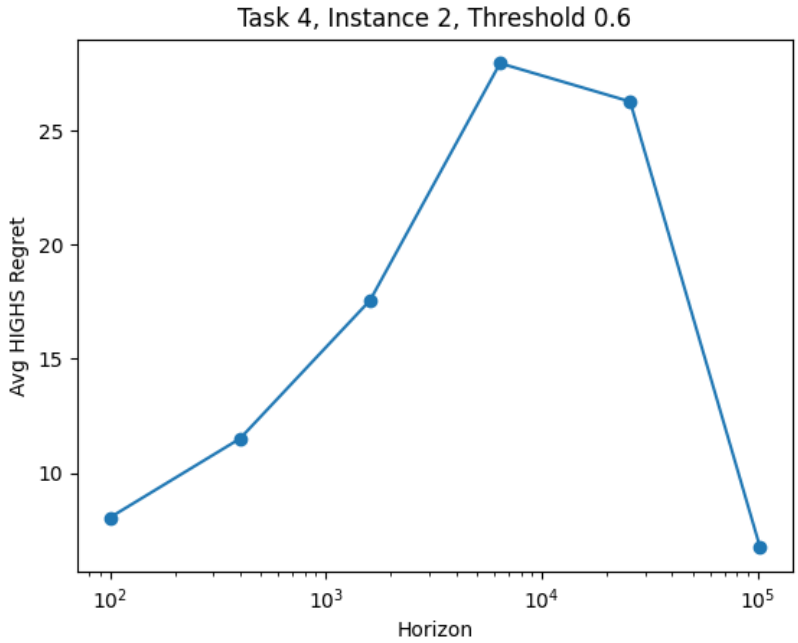
```
      self.dist      = np.array([[float(l) for l in
      ↪  line.strip().split()] for line in lines[1:]])
      self.cumdist   = np.cumsum(self.dist, axis=1)
      self.avg       = np.zeros(self.arms)
      self.count     = np.zeros(self.arms)

    # initialize the methods inherited from Algorithms class
    super().__init__()

  def __call__(self):
    assert self.al == "alg-t4", "incorrect algorithm specified for
    ↪  task 4"
    np.random.seed(self.rs)
    self.thompson()
```

2.4.2 Plots




Task 4, Instance 1, Threshold 0.6
Avg HIGHS Regret
22.5
20.0
17.5
15.0
12.5
10.0
7.5
10^2
10^3
10^4
10^5
Horizon


Task 4, Instance 2, Threshold 0.2
Avg HIGHS Regret
40
35
30
25
20
15
10
5
10^2
10^3
10^4
10^5
Horizon



2.4.3 Inference

We observe that the algorithm suffers very low HIGHS-regret for both instances for both thresholds without any inherent parameter tuning, and is also fairly fast in execution. Thus, I believe this is a very good choice for a large variety of instances to minimise the HIGHS-regret without performing any instance specific optimisation. The trend of regret reducing for a larger horizon is probably due to us pulling the optimal arm and ending up with samples with a larger empirical mean than the theoretical mean of the distribution.