Final Documentation Report

for

BT4014

Analytics Driven Design of Adaptive Systems

Final Project Option (2)

Prepared by:

Teo Hong Kang (A0191178X)

Li Xingxuan (A0198935H)

23 April 2021

Content Page

1. Background Information & Problem Setting

Personalized web services are all working towards adapting their services to users through content and user information. Despite having advances in the past few years, personalized web service still remains very challenging. And it is due to 2 reasons; dynamically changing pools of content and the scale of web services that call for fast learning and computational solutions.

The objective of this project is to identify the most appropriate web-based content at the best time for individual users. With a large amount of news and endless content generation online, it is important that we identify content that best fits the interest of online users.

Figure 1 shows an example of a news pool at one particular timestamp. [F1, F2, F3, F4] are news content generated for users to view. However, F1 has been deemed to be the highlight of the news out of the 4, and has been enlarged as the focal news of the website. In this example, reward = 1 if an user clicks into the story, otherwise 0.

Figure 1: Example news to generate rewards upon click

Traditional algorithms such as epsilon greedy have been widely implemented and are based on the users' response. Another newer method, LinUCB, that also uses articles and user features to make decisions is known as contextual bandit algorithm. A learning algorithm sequentially selects articles to serve users based on contextual information of the user and articles, while simultaneously adapting its article-selection strategy based on user-click feedback to maximize total user clicks in the long run.

2. Data Context

```
['1241852100 109723 0 ',
 'user 2:0.078064 3:0.000444 4:0.726928 5:0.194082 6:0.000482 1:1.000000 ',
 '109723 2:0.333116 3:0.000374 4:0.012356 5:0.374912 6:0.279242 1:1.000000 ',
 '109722 2:0.306008 3:0.000450 4:0.077048 5:0.230439 6:0.386055 1:1.000000 ',
 '109721 2:0.186698 3:0.000002 4:0.395538 5:0.145847 6:0.271915 1:1.000000 ',
 '109720 2:0.140852 3:0.003309 4:0.689836 5:0.032845 6:0.133159 1:1.000000 ',
 '109726 2:0.332495 3:0.000029 4:0.048988 5:0.353874 6:0.264614 1:1.000000 ',
 '109725 2:0.443028 3:0.000059 4:0.020423 5:0.171394 6:0.365097 1:1.000000 ',
 '109728 2:0.362262 3:0.000004 4:0.174201 5:0.267464 6:0.196069 1:1.000000 ',
 '109745 2:0.323432 3:0.000094 4:0.005767 5:0.468827 6:0.201881 1:1.000000 ',
 '109744 2:0.360635 3:0.000071 4:0.065585 5:0.267309 6:0.306400 1:1.000000 ',
 '109747 2:0.239512 3:0.000000 4:0.003650 5:0.612741 6:0.144097 1:1.000000 ',
 '109746 2:0.081244 3:0.000005 4:0.658906 5:0.054825 6:0.205019 1:1.000000 ',
 '109743 2:0.278449 3:0.000002 4:0.012100 5:0.557516 6:0.151933 1:1.000000 ',
 '109742 2:0.428321 3:0.000006 4:0.052761 5:0.350660 6:0.168252 1:1.000000 ',
 '109749 2:0.037081 3:0.000006 4:0.820149 5:0.016605 6:0.126160 1:1.000000 ',
 '109748 2:0.344886 3:0.000001 4:0.139647 5:0.302182 6:0.213284 1:1.000000 ',
 '109667 2:0.361316 3:0.000238 4:0.024162 5:0.349806 6:0.264478 1:1.000000 ',
 '109697 2:0.315874 3:0.000210 4:0.021053 5:0.385211 6:0.277651 1:1.000000 ',
 '109734 2:0.000031 3:0.999526 4:0.000024 5:0.000011 6:0.000407 1:1.000000 ',
 '109735 2:0.306008 3:0.000450 4:0.077048 5:0.230439 6:0.386055 1:1.000000 ',
 '109737 2:0.374700 3:0.000042 4:0.057424 5:0.353131 6:0.214703 1:1.000000 ',
 '109732 2:0.337438 3:0.000018 4:0.049615 5:0.354080 6:0.258849 1:1.000000 ',
 '109711 2:0.282662 3:0.000009 4:0.034199 5:0.417605 6:0.265526 1:1.000000 ',
 '109714 2:0.264355 3:0.000012 4:0.037393 5:0.420649 6:0.277591 1:1.000000\n']
```

Figure 2: data from Verizon Media Labs: Yahoo R6 webscope

Figure 2 shows data downloaded from Yahoo R6 webscope datasets from Verizon Media Labs, between the date 2009.05.09 and 2009.05.10. The data is recorded from one event, with the first index structured as ‘timestamp, STORY, reward’. The second index represents user features (6 dimensions), and is generated through user’s information such as sex and occupation. The subsequent indexes represent article id and several other features, and total up to 6 dimensions as well.

In a total number of 7,388,820 events, we will first do data pre-processing since data is large (Figure 3), and it is time consuming to keep running the data. We have also found 62 unique articles, in which they are stored separately from the events.

```python
import numpy as np
import fileinput
import pickle
from tqdm import tqdm

def generate_data(filenames):

    art_ids = []
    art_feats = []
    events = []

    with fileinput.input(files=filenames) as f:
        for line in tqdm(f):
            if (len(line.split())-10) % 7 != 0:
                # some error data to ignore
                None
            else:
                cols = line.strip().split('|')

                # user data
                user_feat = [float(x[2:]) for x in cols[1].strip().split()[1:]]
                user_click = cols[0].strip().split()[2]

                # article data
                pool_idx = []
                pool_ids = []
                for i in range(2, len(cols)):
                    art_line = cols[i].strip().split()
                    art_id = int(art_line[0])
                    art_feat = [float(x[2:]) for x in art_line[1:]]

                    if art_id not in art_ids:
                        art_ids.append(art_id)
                        art_feats.append(art_feat)

                    pool_idx.append(art_ids.index(art_id))
                    pool_ids.append(art_id)

                # event data
                events.append(
                        [
                            pool_ids.index(int(cols[0].strip().split()[1])),
                            user_click,
                            user_feat,
                            pool_idx
                        ]
                    )


    with open('data/art_feats.pkl', 'wb') as f:
        pickle.dump(art_feats, f)

    with open('data/events.pkl', 'wb') as f:
        pickle.dump(events, f)

if __name__ == '__main__':
    generate_data(('data/ydata-fp-td-clicks-v1_0.20090509',
                   'data/ydata-fp-td-clicks-v1_0.20090510'))
```

Figure 3: Pre-processing codes for data

3. Method Details

We will be incorporating and comparing algorithms taught in class, as well as exploring a contextual based model, known as LinUCB for this project. We adopted a learning rate, where by default 90% of data will be used for learning, and the remaining 10% (also known as deployment data) will be used to test the performance of the algorithm. The algorithms that we have explored for this project are: Epsilon Greedy, Epsilon Decay, Annealing Softmax, UCB, Bayesian UCB, LinUCB.

3.1 Epsilon Greedy

We implement the same epsilon greedy algorithm on the dataset. The number of arms is the total number of articles and the selection of both exploitation and exploration is from a pool of articles, which is a subset of total articles. The variables user_feat and art_feat are not used here, only used in LinUCB. With a probability of epsilon, the algo will recommend a random article, otherwise, the article with maximum reward will be recommended.

```
class EpsilonGreedy():

    def __init__(self, epsilon, n_arms):
        self.epsilon = epsilon
        self.counts = np.zeros(n_arms)
        self.values = np.zeros(n_arms)
        self.algo = 'EpsilonGreedy (ε=' + str(epsilon) + ')'

    def choose_arm(self, n_trails, user_feat, pool_idx, art_feat):
        """

        Return chosen arm relative to the pool

        Arguments:
            n_trails {int} -- number of trails
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool

        """

        if np.random.rand() > self.epsilon:
            return np.argmax(self.values[pool_idx])
        else:
            return np.random.randint(low=0, high=len(pool_idx))

    def update(self, chosen_arm, reward, user_feat, pool_idx, art_feat):
        """

        Update parameters of the algo

        Arguments:
            chosen_arm {int} -- chosen article index relative to the pool
            reward {int} -- binary, user click is 1, not click is 0
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool
        """

        a = pool_idx[chosen_arm]
        self.counts[a] += 1
        n = self.counts[a]
        self.values[a] = ((n-1)/float(n))*self.values[a] + (1/float(n))*reward
```

Figure 4: Epsilon Greedy Algorithm

3.2 Epsilon Decay

We also follow the same epsilon decay algorithm introduced in class. The epsilon decays when the number of trials increases.

```
class EpsilonDecay():

    def __init__(self, n_arms):
        self.counts = np.zeros(n_arms)
        self.values = np.zeros(n_arms)
        self.algo = 'EpsilonDecay'

    def choose_arm(self, n_trails, user_feat, pool_idx, art_feat):
        """

        Return chosen arm relative to the pool

        Arguments:
            n_trails {int} -- number of trails
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool

        """

        if np.random.rand() > 1/(sum(self.counts)/len(self.counts)+1):
            return np.argmax(self.values[pool_idx])
        else:
            return np.random.randint(low=0, high=len(pool_idx))

    def update(self, chosen_arm, reward, user_feat, pool_idx, art_feat):
        """

        Update parameters of the algo

        Arguments:
            chosen_arm {int} -- chosen article index relative to the pool
            reward {int} -- binary, user click is 1, not click is 0
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool
        """

        a = pool_idx[chosen_arm]
        self.counts[a] += 1
        n = self.counts[a]
        self.values[a] = ((n-1)/float(n))*self.values[a] + (1/float(n))*reward
```

Figure 5: Epsilon Decay Algorithm

3.3 Annealing Softmax

For the temperature of the annealing softmax, where each arm is assigned a probability to be chosen. We follow the implementation of the class, where $temperataure = \frac{1}{1+log(n+0.000001)}$.

```python
class AnnealingSoftmax():

    def __init__(self, n_arms):
        self.counts = np.zeros(n_arms)
        self.values = np.zeros(n_arms)
        self.algo = 'AnnealingSoftmax'

    def choose_arm(self, n_trails, user_feat, pool_idx, art_feat):
        """

        Return chosen arm relative to the pool

        Arguments:
            n_trails {int} -- number of trails
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool

        """

        temperature = 1/(1+np.log(sum(self.counts[pool_idx])+0.000001))
        z=sum([np.exp(v/temperature) for v in self.values[pool_idx]])
        probs=[np.exp(v/temperature)/z for v in self.values[pool_idx]]
        return np.random.choice(len(pool_idx), p=probs)

    def update(self, chosen_arm, reward, user_feat, pool_idx, art_feat):
        """

        Update parameters of the algo

        Arguments:
            chosen_arm {int} -- chosen article index relative to the pool
            reward {int} -- binary, user click is 1, not click is 0
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool
        """

        a = pool_idx[chosen_arm]
        self.counts[a] += 1
        n = self.counts[a]
        self.values[a] = ((n-1)/float(n))*self.values[a] + (1/float(n))*reward
```

Figure 6: Annealing Softmax Algorithm

3.4 UCB

Similar to the algorithm in class, we implement the UCB algorithm and generalize the confidence level to a hyperparameter alpha.

```
class UCB1():

    def __init__(self, n_arms, alpha):
        self.counts = np.zeros(n_arms)
        self.values = np.zeros(n_arms)
        self.alpha = alpha
        self.algo = 'UCB1 (α=' + str(alpha) + ')'

    def choose_arm(self, n_trails, user_feat, pool_idx, art_feat):
        """

        Return chosen arm relative to the pool

        Arguments:
            n_trails {int} -- number of trails
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool

        """

        ucb_values = self.values[pool_idx] + \
                     np.sqrt(self.alpha * np.log(n_trails + 1) / self.counts[pool_idx])
        return np.argmax(ucb_values)

    def update(self, chosen_arm, reward, user_feat, pool_idx, art_feat):
        """

        Update parameters of the algo

        Arguments:
            chosen_arm {int} -- chosen article index relative to the pool
            reward {int} -- binary, user click is 1, not click is 0
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool
        """

        a = pool_idx[chosen_arm]
        self.counts[a] += 1
        n = self.counts[a]
        self.values[a] = ((n-1)/float(n))*self.values[a] + (1/float(n))*reward
```

Figure 7: UCB Algorithm

3.5 Bayesian UCB

Since BayesUCB takes a lot of time to run, we only run one set of hyperparameters. Where the number of standard deviations is 3 and both initial alpha and beta are 1.

```
class BayesUCB():

    def __init__(self, n_arms, stdnum=3, init_alpha=1, init_beta=1):
        self.counts = np.zeros(n_arms)
        self.values = np.zeros(n_arms)
        self.alphas = np.array([init_alpha] * n_arms)
        self.betas = np.array([init_beta] * n_arms)
        self.stdnum = stdnum
        self.algo = 'Baysian UCB'

    def choose_arm(self, n_trails, user_feat, pool_idx, art_feat):
        """

        Return chosen arm relative to the pool

        Arguments:
            n_trails {int} -- number of trails
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool

        """

        pool_alphas = self.alphas[pool_idx]
        pool_betas = self.betas[pool_idx]

        best_arm = max(
                range(len(pool_idx)),
                key=lambda x: pool_alphas[x] / float(pool_alphas[x] + pool_betas[x]) + \
                    beta.std(
                        pool_alphas[x], pool_betas[x]
                    ) * self.stdnum
            )
        return best_arm

    def update(self, chosen_arm, reward, user_feat, pool_idx, art_feat):
        """

        Update parameters of the algo

        Arguments:
            chosen_arm {int} -- chosen article index relative to the pool
            reward {int} -- binary, user click is 1, not click is 0
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool
        """

        a = pool_idx[chosen_arm]
        self.counts[a] += 1
        n = self.counts[a]
        self.values[a] = ((n-1)/float(n))*self.values[a] + (1/float(n))*reward
        self.alphas[a] += reward
        self.betas[a] += (1-reward)
```

Figure 8: BayesianUCB Algorithm

3.6 LinUCB

All of the above algorithms are gaining historical rewards vertically without features of users and articles. Thus, we incorporated a contextual-bandit algorithm, known as LinUCB. It observes the current user and a set of articles with their feature vectors. The vector summarizes information of both the user and articles and will be referred to as the context. Based on observed rewards in previous trials, the algorithm chooses an article and receives payoff, whose expectation depends on both the user and the arm. The algorithm then improves article-selection strategy with the new observation.

Algorithm 1 LinUCB with disjoint linear models.

Inputs: $\alpha \in \mathbb{R}_+$
for $t = 1, 2, 3, \ldots, T$ **do**
  Observe features of all arms $a \in \mathcal{A}_t$: $\mathbf{x}_{t,a} \in \mathbb{R}^d$
  for all $a \in \mathcal{A}_t$ **do**
    if a is new **then**
      $\mathbf{A}_a \leftarrow \mathbf{I}_d$ (d-dimensional identity matrix)
      $\mathbf{b}_a \leftarrow \mathbf{0}_{d\times 1}$ (d-dimensional zero vector)
    end if
    $\hat{\boldsymbol{\theta}}_a \leftarrow \mathbf{A}_a^{-1}\mathbf{b}_a$
    $p_{t,a} \leftarrow \hat{\boldsymbol{\theta}}_a^\top \mathbf{x}_{t,a} + \alpha\sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1}\mathbf{x}_{t,a}}$
  end for
  Choose arm $a_t = \arg\max_{a\in\mathcal{A}_t} p_{t,a}$ with ties broken arbitrarily, and observe a real-valued payoff r_t
  $\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + \mathbf{x}_{t,a_t}\mathbf{x}_{t,a_t}^\top$
  $\mathbf{b}_{a_t} \leftarrow \mathbf{b}_{a_t} + r_t\mathbf{x}_{t,a_t}$
end for

Figure 9: Pseudocode for LinUCB Algorithm

```python
class LinUCB():

    def __init__(self, n_arms, alpha):

        # size for A, b matrices is 6*2=12
        d = 12
        self.A = np.array([np.identity(d)] * n_arms)
        self.b = np.zeros((n_arms, d, 1))
        self.alpha = alpha
        self.algo = 'LinUCB (α=' + str(alpha) + ')'

    def choose_arm(self, n_trails, user_feat, pool_idx, art_feat):
        """

        Return chosen arm relative to the pool

        Arguments:
            n_trails {int} -- number of trails
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool

        """

        A = self.A[pool_idx]
        b = self.b[pool_idx]
        user = np.array([user_feat] * len(pool_idx))
        art_feat = np.array(art_feat)

        A = np.linalg.inv(A)
        x = np.hstack((user, art_feat[pool_idx]))

        x = x.reshape((len(pool_idx), 12, 1))

        theta = A @ b
        p = np.transpose(theta, (0, 2, 1)) @ x + self.alpha * np.sqrt(
            np.transpose(x, (0, 2, 1)) @ A @ x
            )
        return np.argmax(p)

    def update(self, chosen_arm, reward, user_feat, pool_idx, art_feat):
        """

        Update parameters of the algo

        Arguments:
            chosen_arm {int} -- chosen article index relative to the pool
            reward {int} -- binary, user click is 1, not click is 0
            user_feat {np.array} -- user feature arrary (1,6)
            pool_idx {np.array} -- indexes for article pool
        """

        a = pool_idx[chosen_arm]
        x = np.hstack((user_feat, art_feat[a]))
        x = x.reshape((12, 1))

        self.A[a] = self.A[a] +  x @ np.transpose(x)
        self.b[a] += reward * x
```

Figure 10: LinUCB Algorithm

4. Performance Analysis

In Figure 11 we have provided the policy evaluator pseudocode that our results will be evaluated based on.

Algorithm 2 `Policy_Evaluator` (with finite data stream).

bandit algorithm A; stream of events S of length L
$h_0 \leftarrow \emptyset$ {An initially empty history}
$\hat{G}_{\mathsf{A}} \leftarrow 0$ {An initially zero total payoff}
$T \leftarrow 0$ {An initially zero counter of valid events}
for $t = 1, 2, 3, \ldots, L$ **do**
  Get the t-th event $(\mathbf{x}, a, r_a)$ from S
  if $\mathsf{A}(h_{t-1}, \mathbf{x}) = a$ **then**
    $h_t \leftarrow \text{CONCATENATE}(h_{t-1}, (\mathbf{x}, a, r_a))$
    $\hat{G}_{\mathsf{A}} \leftarrow \hat{G}_{\mathsf{A}} + r_a$
    $T \leftarrow T + 1$
  else
    $h_t \leftarrow h_{t-1}$
  end if
end for
Output: $\hat{G}_{\mathsf{A}}/T$

Figure 11: Pseudocode for Policy Evaluator

On the evaluation policy, we adopt an unbiased offline evaluation algorithm from the paper "Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms" *(Li, L., Chu, W., Langford, J. and Wang, X., 2012)*. Running through the events, only when the article selected at the event is the same as the article selected by the algorithm, we will update the reward and the algorithm, else we ignore the event. And in the implementation, we also randomly split a portion of data as a testing dataset. On these events, the algorithm will also not be updated. Also, we run a completely randomly selected algorithm on all events as a baseline. We then calculate the ratio between the algorithm's cumulative result and the baseline model to compare the improvement. The pseudocode algorithm is shown in Figure 12.

```python
def test_algo(algo, events, art_feats, size_rate=None, learn_rate=0.9):

    start = time.time()
    G_learn = 0
    G_deploy = 0
    N_learn = 0
    N_deploy = 1

    exp_learns = []
    exp_deploy = []

    if size_rate is None:
        events = events
    else:
        events = random.sample(events, int(len(events)*size_rate/100))

    for i, event in enumerate(tqdm(events)):
    # for i, event in enumerate(events):
        dis = event[0]
        reward = int(event[1])
        user_feat = event[2]
        pool_idx = event[3]

        chosen_art = algo.choose_arm(N_learn+N_deploy, user_feat, pool_idx, art_feats)
        if chosen_art == dis:
            # update only when chosen is displayed
            if random.random() < learn_rate:
                # update with learn rate
                G_learn += reward
                N_learn += 1
                algo.update(dis, reward, user_feat, pool_idx, art_feats)
                exp_learns.append(G_learn/N_learn)

            else:
                # dont update
                G_deploy += reward
                N_deploy += 1
                exp_deploy.append(G_deploy/N_deploy)

    end = time.time()

    exc_time = round(end-start, 1)
    print(algo.algo, round(G_deploy/N_deploy, 4), exc_time)

    return exp_learns, exp_deploy
```

Figure 12: Testing Algorithm's Code

4.1 Result

Upon implementation of all the respective algorithms and following the test_algo method, we have derived the resulting graphs plotted by CTR against T. CTR refers to the Click-Through-Rate of the respective websites, whereas T refers to each successive time period.

4.1.1 Epsilon Greedy

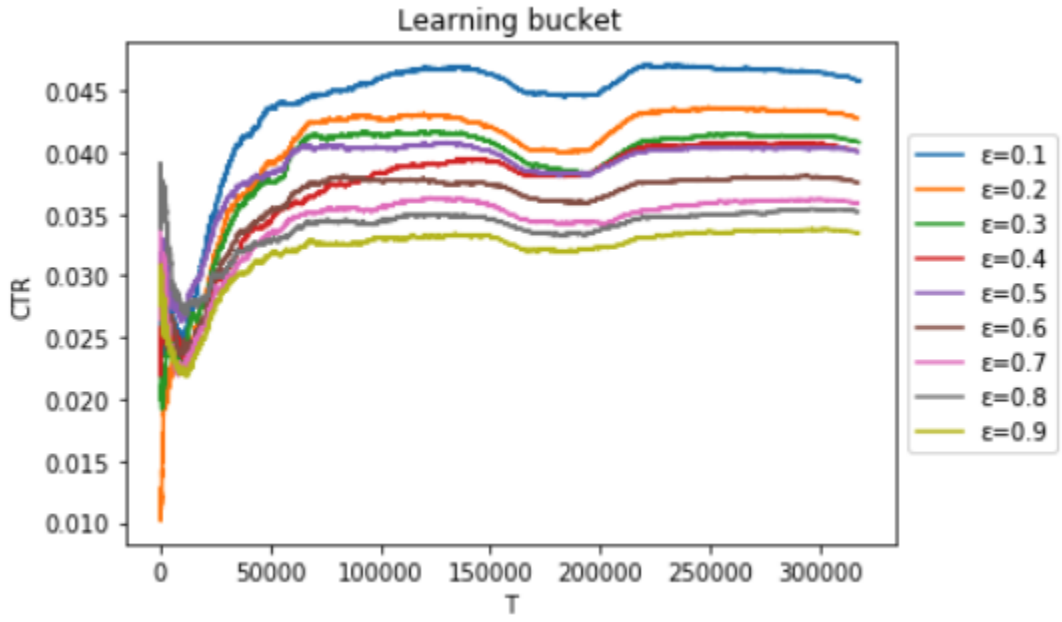

Figure 13: Epsilon Greedy's Result

We run a range of epsilons to compare the performance of the epsilon greedy algorithm. And we notice that when epsilon is 0.1, it gives the best performance in the long run. And it concaves at around 175000 trials. We think that the reason is because at that certain time period, maybe there is a break news, and users who do not usually read those content will also be attracted by the news. Thus our algorithm cannot capture that.

4.1.2 Epsilon Decay

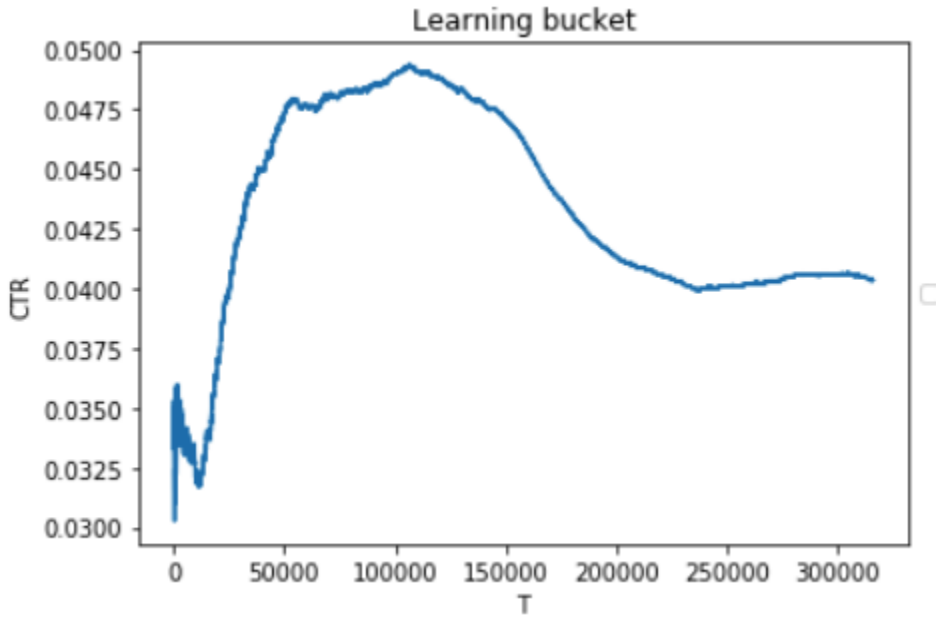

Figure 14: Epsilon Decay's Result

For the epsilon decay, we notice that there is a significant draw down starting from 100000 trails. We think that the reason is that at the 100000th trial, epsilon equals 10e-10 which exploration of the algorithm can be neglected. And when there are new articles coming in and old articles stop showing on the website, the algorithm will always give wrong recommendations, which explains the draw down.

4.1.3 Annealing Softmax

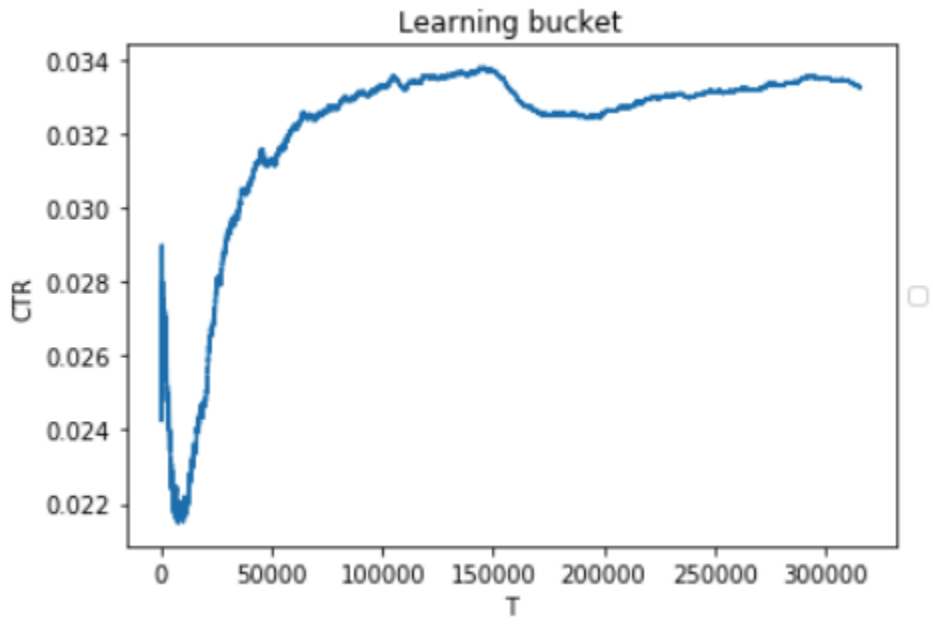

Figure 15: Annealing Softmax's Result

Similar concave area happens in Annealing Softmax. We believe it is the same reason; when there are new articles coming in and old articles stop showing on the website, the algorithm will always give wrong recommendations, which explains the draw down.

4.1.4 UCB

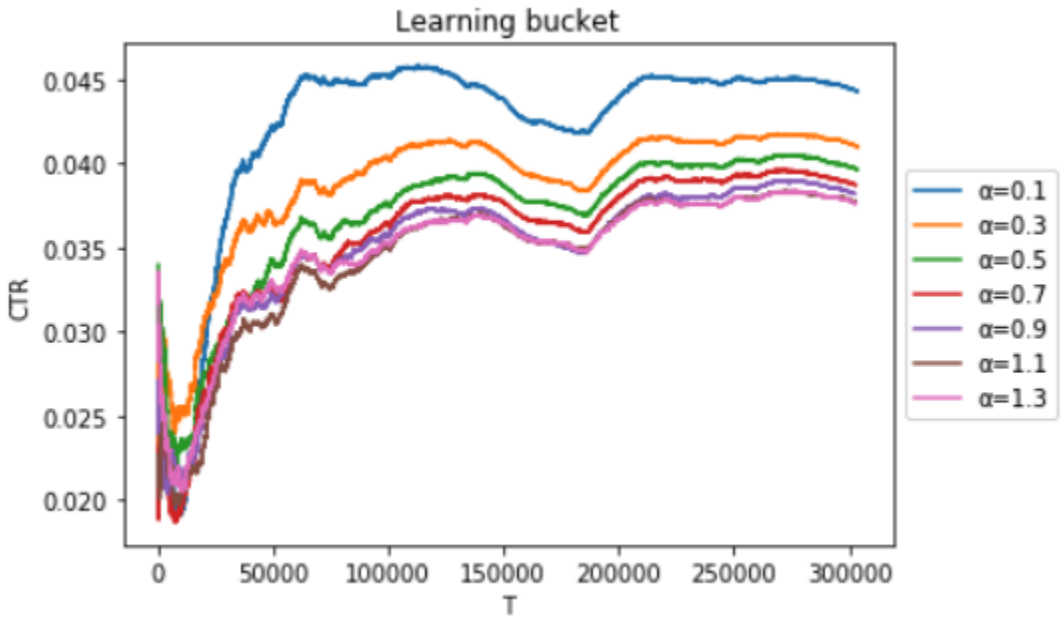

Figure 16: UCB's Result

When the confidence level is 0.1, it gives the best result. Confidence level controls the level of exploration. Thus, least exploration leads to the best result; as the confidence level increases, the CTR value slowly decreases respectively.

4.1.5 BayesUCB

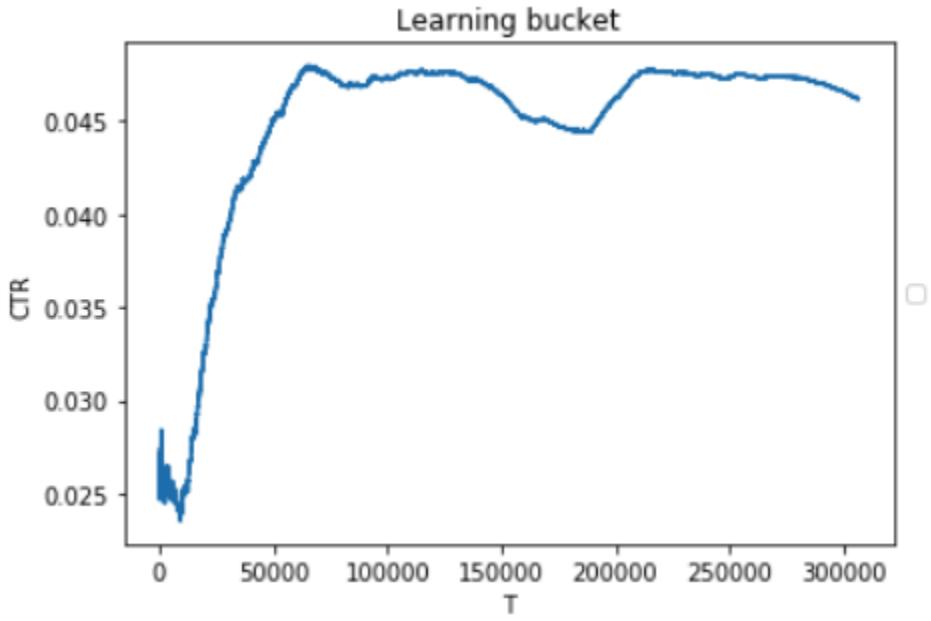

Figure 17: BayesUCB's Result

In regards to runtime, BayesUCB takes 7.5 hours to finish one run. Therefore, we only conduct a test with the number of standard deviations of 3.

4.1.6 LinUCB

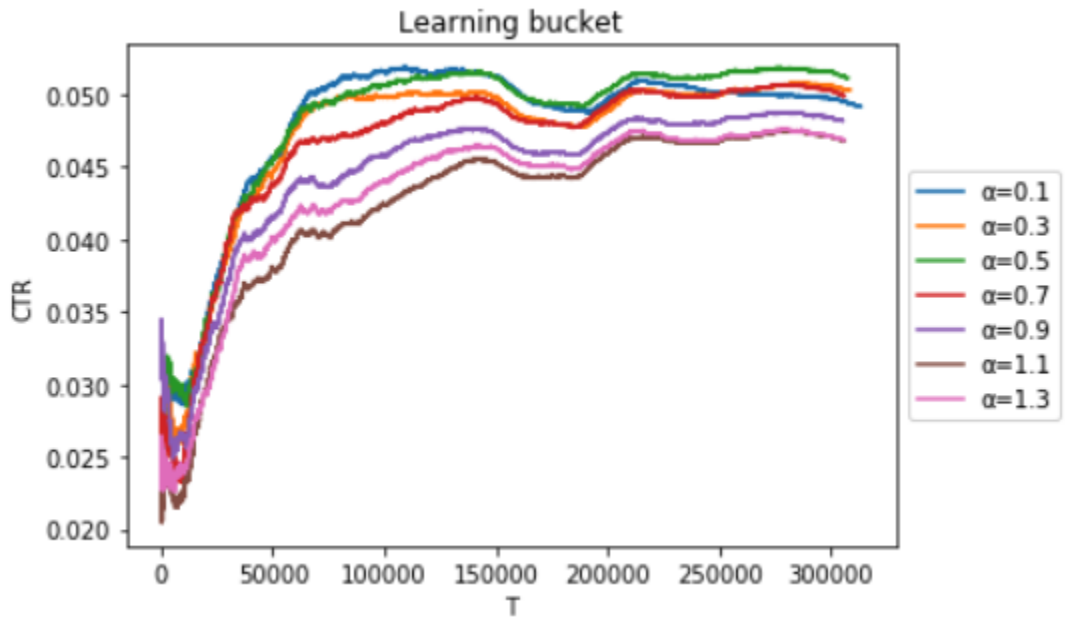

Figure 18: LinUCB's Result

As shown in the graph, in the long run, the best CTR occurs when alpha=0.5. However in the short run, apha equals 0.1 gives a better result. More specifically, when T is between 50000-100000. In our opinion, this is because when new similar articles come in, the algorithm will need more exploration to select those new articles with similar features. But in the short run, when there are not many new articles, sticking with exploitation may be a better choice.

4.2 Comparing all models with best parameters

4.2.1 Learning bucket

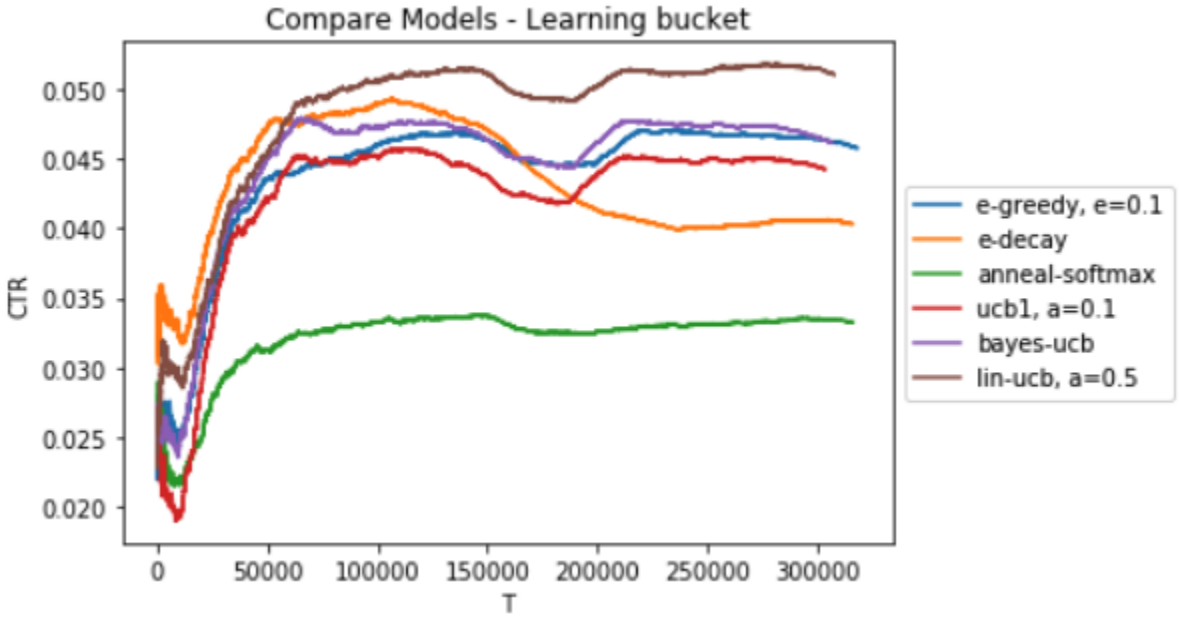

Figure 19: Learning Bucket Graph for all models

LinUCB provides the highest CTR value in the long run, whereas Annealing Softmax provides the least CTR value in the long run.

4.2.2 Deploy bucket

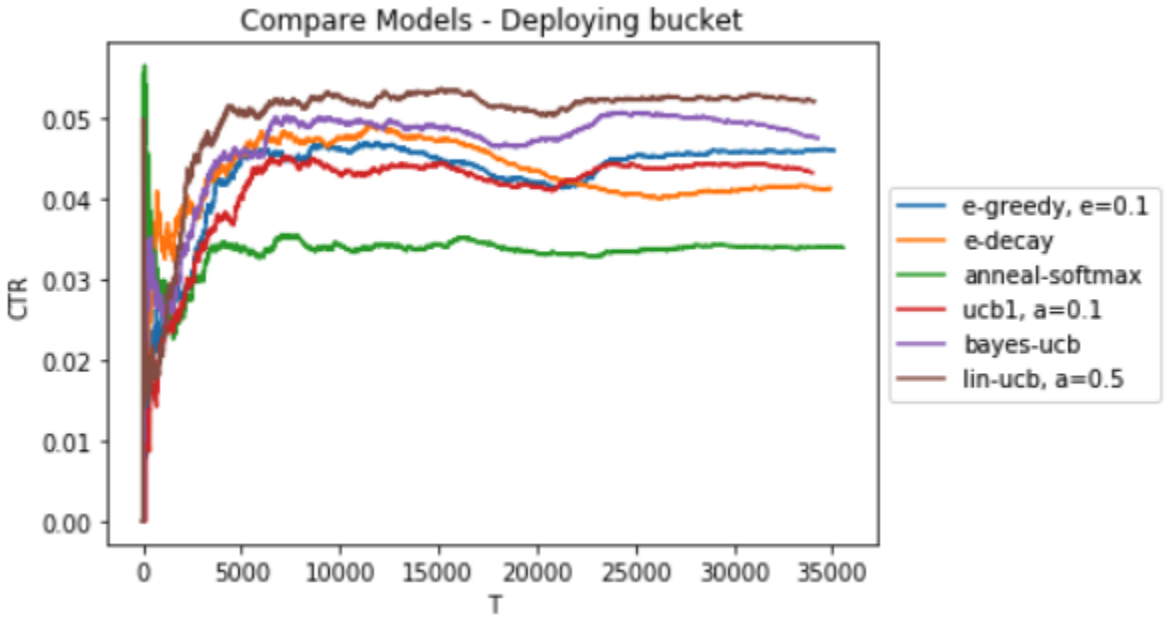

Figure 20: Deploy Bucket Graph for all models

LinUCB provides the highest CTR value in the long run, whereas Annealing Softmax provides the lowest CTR value in the long run.

4.2.3 Compare with random model

4.2.3.1 Ratio with random model (Epsilon=1, CTR=0.0331)

We first run a completely random bandit algorithm, which means at each timestamp, the article is completely selected randomly. And CTR = 0.0331. Then, we calculate the ratio between the best performance of each model and the random model to see the improvement.

Best Model	Learning bucket	Deploy bucket
e-greedy, e=0.1	1.46	1.39
e-decay	1.29	1.25
anneal-softmax	1.06	1.03
ucb1, a=0.1	1.41	1.31
bayes-ucb	1.48	1.44
lin-ucb, a=0.5	1.63	1.58

5 Other Thoughts

5.1 Difficulties of project

The dataset that we have worked on is very large, and in order for us to carry out subsequent steps of the project, a lot of pre-processing had to be done. We have also identified 62 unique articles and had to store these key information, making it harder to effectively pre-process the data that we had.

On the other hand, when it comes to evaluating the performance of each model, we had to take into account all different parameter values which may lead to different resulting graphs. It is important to explore every possible parameter value to ensure that we find the best fit result. After obtaining the best results from each successive model, we also had to evaluate the best model among all.

Lastly, the dataset that we worked on is given in the form of embedded feature vectors. Hence, we are unable to sift out meaningful features for descriptive analysis. This restricts us from gathering further background information on the population details of the users that accessed the website.

6 Conclusion

In this project, we explored using both traditional bandit algorithms as well as contextual bandit algorithms to make recommendations for news articles on real world yahoo dataset. We found out that for contextual bandit algorithms, it can learn features of items and users and recommend items even if they are not inside the article pool, which is very useful for recommendation systems such as recommending new movies. To explore further all of the testing algorithms, we can test them on many articles to compare performances as well. Lastly, contextual bandit algorithms are not only limited to the online news industry, but can also be extended to other different industries, such as financial portfolio constructions. This can be done by replacing article feature vectors with stock feature vectors, or possibly working with datasets of different structure.

7 Reference

Li, L., Chu, W., Langford, J. and Wang, X., 2012. *Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms*. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1003.5956.pdf> [Accessed 19 April 2021].