NUS National University of Singapore | School of Computing

Bandit Algorithm Application on Real-world Problem

Teo Hong Kang - A0191178X
Li Xing Xuan - A0198935H

Summary

01
Background Information

02
Problem Setting

03
Data Context

04
Method Details

05
Performance Analysis

06
Difficulties

07
Conclusion

Background Information

- Personalized Web Services adapting services to users
  - Content
  - User Information

- Personalized Web Services remain challenging despite years of advancement
  - Dynamically changing pools of content
  - Scale of web services

Problem Setting

Objective

- Identify the most appropriate web-based content generation online
- Important to identify content that best fits the interest of online users

Example

Problem Setting

- Traditional Algorithms (Eg. Epsilon Greedy)
  - Widely implemented
  - Based on user's response

- Newer Algorithms (LinUCB)
  - Based on articles and user features
  - Known as Contextual Bandit Algorithm

Data Context

- ➔ Data from Yahoo R6 webscope datasets - Verizon Media Labs
- ➔ Recorded from 1 event - 2009.05.09 - 2009.05.10
- ➔ First index structure
  - ◆ ‘Timestamp, STORY, reward”
- ➔ Second idex structure
  - ◆ User features
- ➔ Subsequent index structure
  - ◆ Article ID + other features

First index

Second index

```
['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']
```

Subsequent index

Data Context

- ➔ Total number of 7,388,820 events
- ➔ Conduct data pre-processing
  - ◆ Data is large
  - ◆ Time consuming to keep running data
- ➔ 62 unique articles, 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'))
```

Method Details

- Incorporate & compare algorithms taught in class
- Explore contextual based model
  - LinUCB
- Learning rate
  - 90% of data for learning
  - 10% of data for deployment

Algorithms Explored

1. Epsilon Greedy
2. Epsilon Decay
3. Annealing Softmax
4. UCB
5. Bayesian UCB
6. LinUCB

Algorithm 1: Epsilon Greedy

- Number of Arms = Total number of Articles
- Selection of both Exploitation & Exploration is from a pool of articles, subset of total articles

```python
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
```

Algorithm 2: Epsilon Decay

- ➔ Follows similar algorithm taught in class; epsilon decays when number of trials increases

```python
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 array (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 array (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
```

Algorithm 3: Annealing Softmax

➔ Temperature of Annealing Softmax follows implementation taught in class, where

$$temperature = \frac{1}{1+log(n+0.000001)}$$

```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
```

Algorithm 4: UCB

- ➔ Similar to algorithm taught in class
- ➔ Confidence level generalized to a hyperparameter - alpha

```python
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
```

Algorithm 5: Bayesian UCB

- Takes a long time to run
  - Only run one set of hyperparameters
  - Standard Deviation = 3
  - Initial Alpha = 1
  - Initial Beta = 1

```python
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)
```

Algorithm 6: LinUCB

- ➔ All of the previously mentioned algorithms gain Historical Rewards vertically
  - ◆ Absence of User Features & Articles
- ➔ Implement Contextual-bandit Algorithm: LinUCB
  - ◆ Observes the current user + set of articles with its feature vectors
- ➔ Based on observed rewards in previous trials
  - ◆ Algorithm chooses an article
  - ◆ Receives Payoff (dependent of user & arm)
- ➔ Improves Article-selection strategy with 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

Algorithm 6: LinUCB

Algorithm's Code

```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
```

Performance Analysis

Policy Evaluator Pseudocode for results to be evaluated 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**
        $h_t \leftarrow \textsc{concatenate}(h_{t-1}, (\mathbf{x}, a, r_a))$
        $\hat{G}_{\mathsf{A}} \leftarrow \hat{G}_{\mathsf{A}} + r_a$
        $T \leftarrow T + 1$
    else
        $h_t \leftarrow h_{t-1}$
    end if
end for
Output: $\hat{G}_{\mathsf{A}}/T$

- ➔ Adopted unbiased offline evaluation algorithm from “Unbiased Offline Evaluation of Contextual-bandit-based” article
- ➔ When article selected at event = article selected by algorithm
  - ◆ Update reward
  - ◆ Update algorithm
  - ◆ Else, event is ignored

Performance Analysis

- ➔ Randomly split a portion of data as test dataset
  - ◆ Algorithms will not be updated for these events
- ➔ Completely random selected algorithm on all events as a baseline
- ➔ Compare improvement
  - ◆ Calculate ratio between algorithm's Cumulative Result and Baseline Model

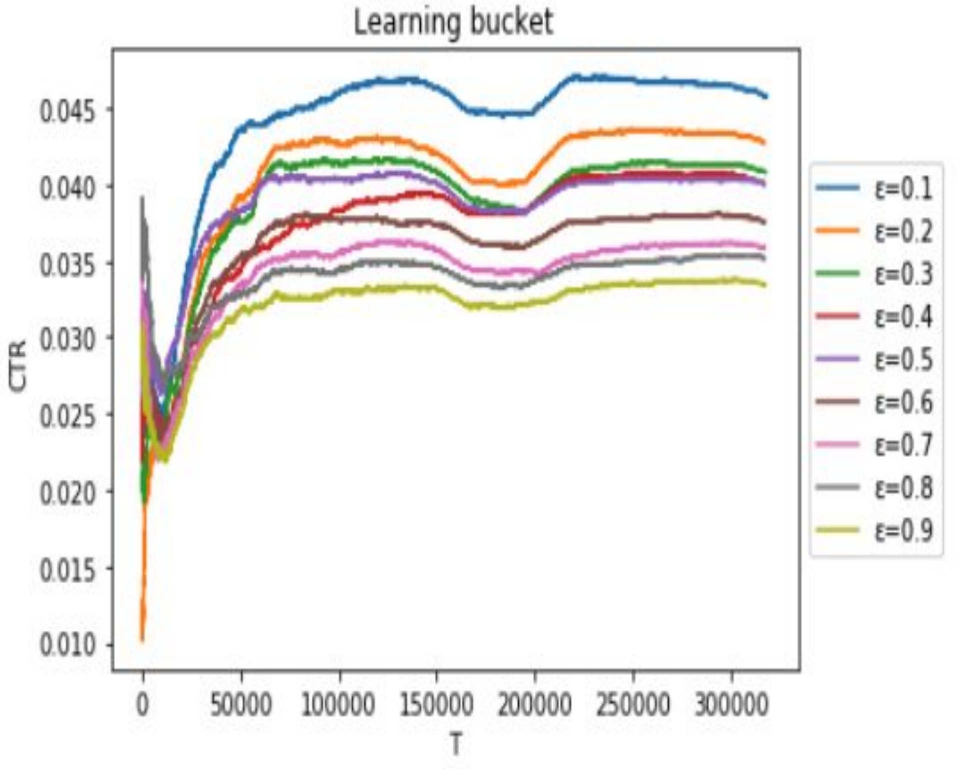
```python
def test_algo(algo, events, art_feats, size_rate=None, learn_rate=0.9):

    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
```

Results - Epsilon Greedy

- ➔ Run a range of Epsilon value to compare performance
- ➔ Best performance in the long run when Epsilon = 0.1
- ➔ Concaves at ~175,000 trials

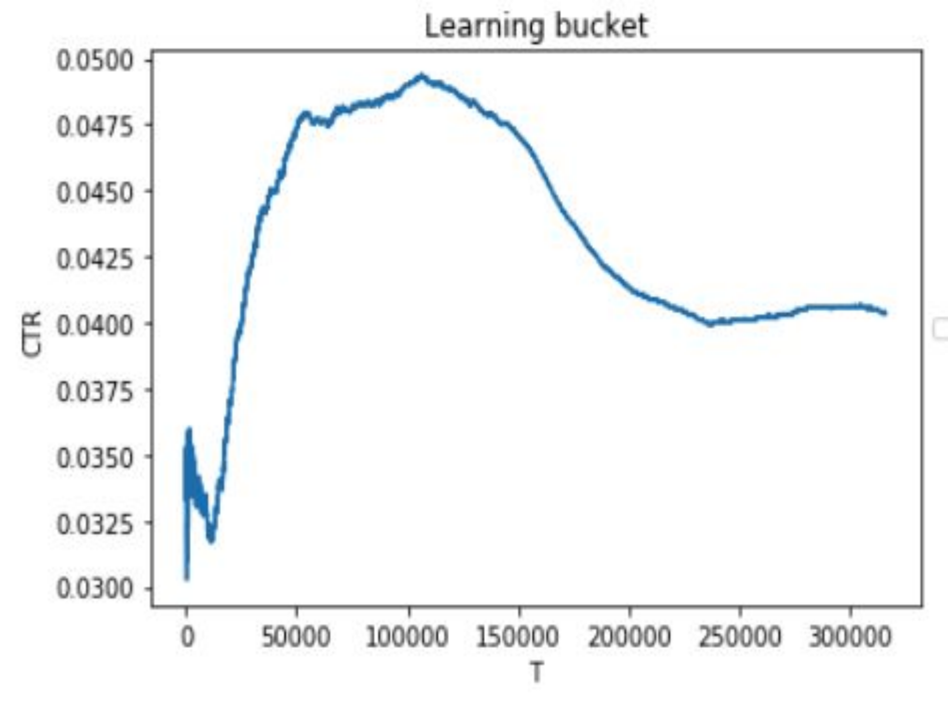
Possible Reason of Concave:

- Breaking News at certain time period
- Many other users will be attracted
- Algorithm doesn’t capture



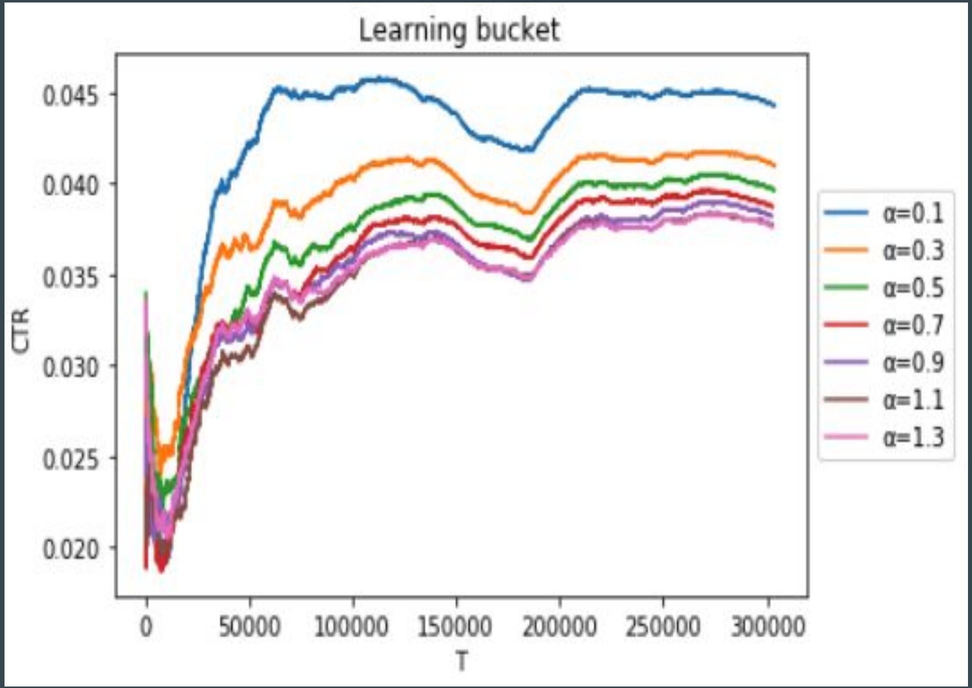
Results - Epsilon Decay

- Significant draw down starting from 100,000 trials
  - Probable reason: epsilon = 10e^-10, exploration of algorithm neglected
  - New articles appear on website; old articles stop showing on website
    - Algorithm gives wrong recommendation



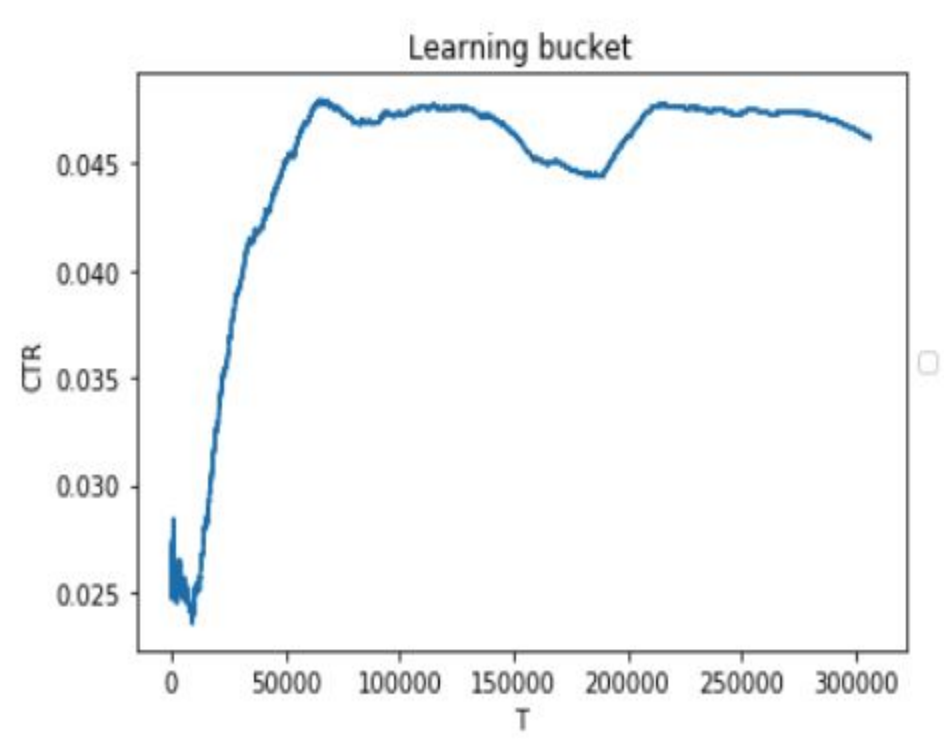
Results - UCB

- ➔ When Confidence Level = 0.1
  - ◆ Provides best result
- ➔ Confidence Level controls level of exploration
  - ◆ Least exploration = Best Result



Results - BayesUCB

- ➔ With regards to run time, BayesUCB takes 7.5hrs to complete one run
- ➔ Only conduct a test with Standard Deviation = 3

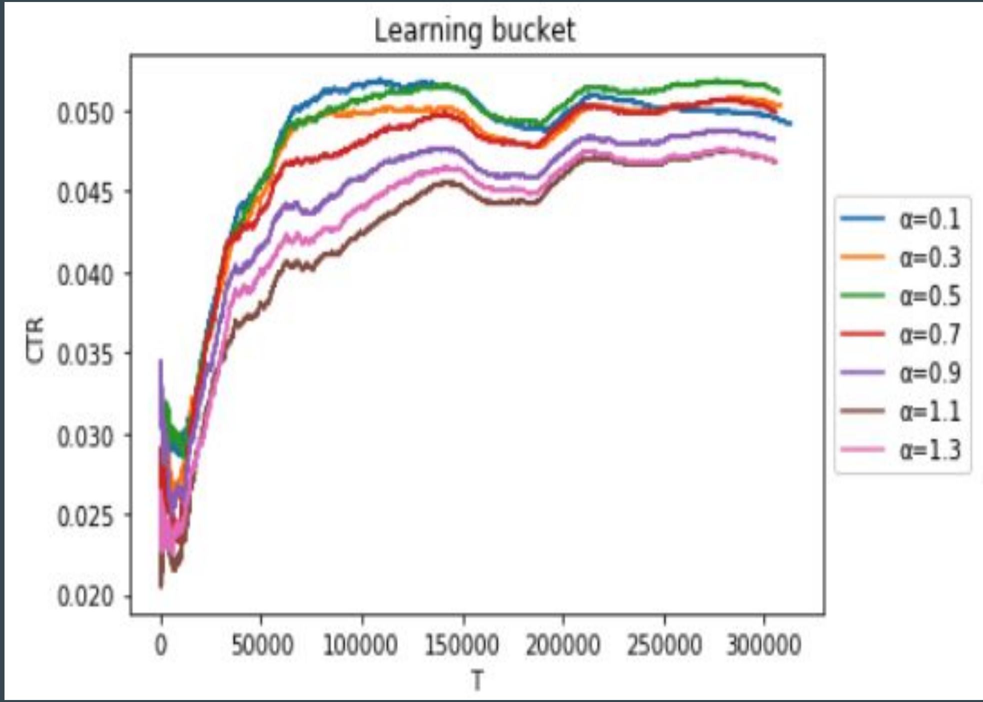


Results - LinUCB

- ➔ Best CTR in the long run: Alpha = 0.5
- ➔ Best CTR in the short run: Alpha = 0.1

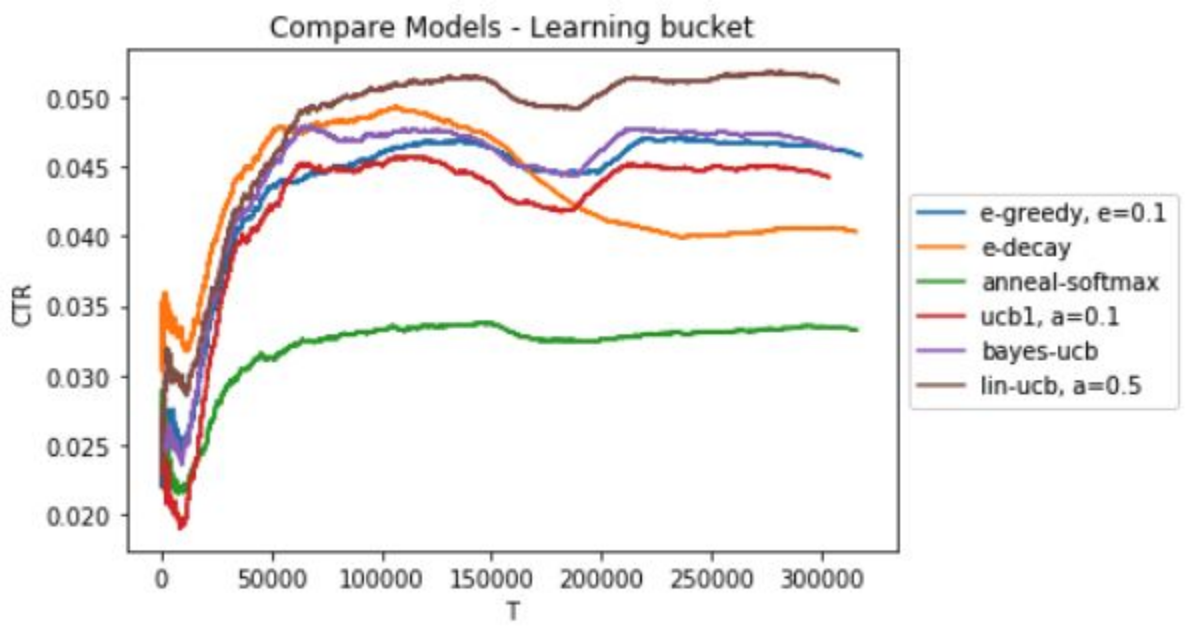
Possible Reason:

- In the long run
  - New similar articles appear
  - Algorithm require more exploration to select new articles of similar feature
- In the short run
  - Not many new articles
  - Sticking with exploitation is a better choice



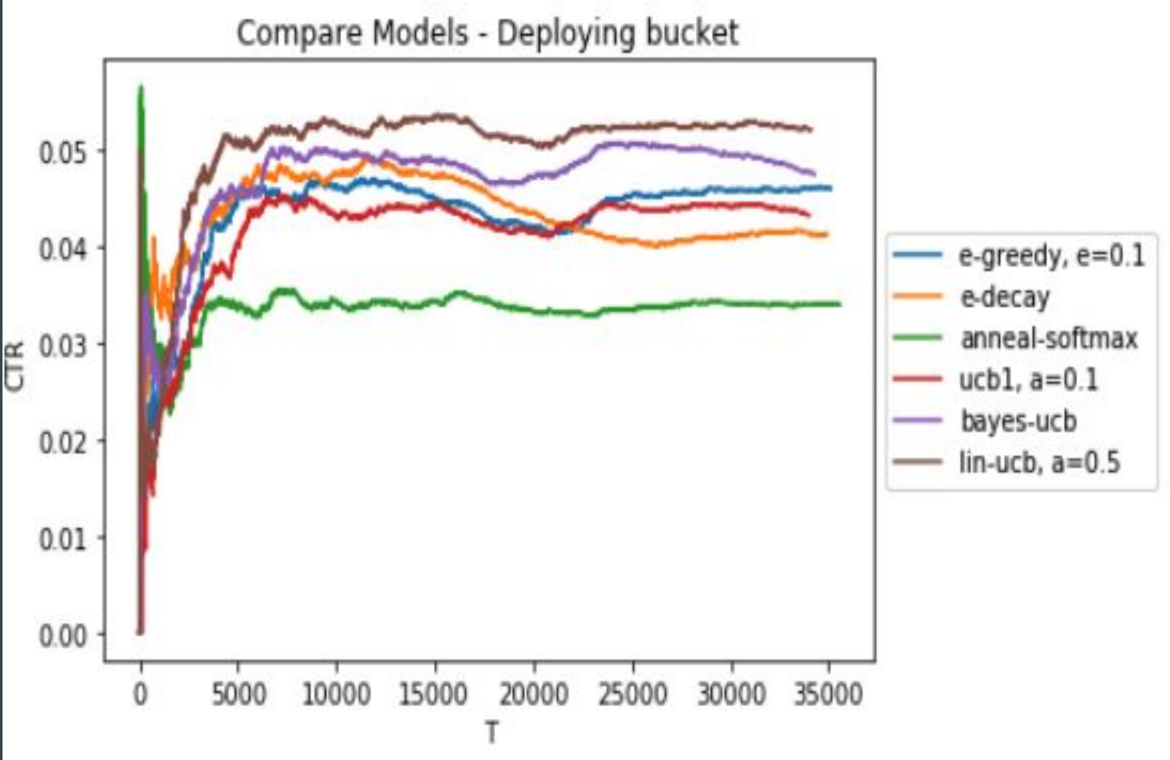
Results - Comparing all models with Best Parameters

Learning Bucket



Results - Comparing all models with Best Parameters

Deploy Bucket



Results - Ratio with Random Model (Epsilon=1, CTR=0.0331)

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

Difficulties

1. Huge dataset:
    a. Many pre-processing steps for the dataset
    b. Store key information
2. Different performances of same model
    a. With different parameters, the performance of the same model can be different
    b. Had to experiment with many different parameter values to find the best result
3. Cannot perform descriptive analysis
    a. The data is given in the form of embedding feature vectors
    b. No meaningful features can be used for descriptive analysis

Conclusion

1. Contextual bandit algorithm can learn features of items and users
   - It can recommends items even if they are not in the article pool
   - useful for recommendation system (e.g. new movies)
2. To explore further, we can test on the number of articles to compare performances
3. Contextual bandit algorithm can be extended to many other industries (e.g. financial portfolio construction)

References

1. 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].