Contextual Bandits for Personalized News Article Recommendation

Kilol Gupta (kg2719)[1]

[1]Columbia University, SEAS

kilol.gupta@columbia.edu

Abstract. *The report will outline the specific details of this assignment which aimed at building a personalized news article recommendation system using Contextual Bandits approach on a per-sonalization data-set. Contextual bandits is usually an online recommendation algorithm but since our data-set is not interactive in nature, we had to treat the offline nature of data-set in an offline fashion and use offline evaluation tech-niques such as cumulative take rate (replay menthod). Three strategies to pick alpha (the hyperparameter controlling the confidence bound) have been imple-mented to find an optimal value for it.*

Index

1. Literature Review

Contextual bandits are a popular approach to problems like personalized medical treatment or curation of online content on various social media websites. The specific type of bandits formulation that we are exploring via this assignment involve the k-armed bandits problem which has a fixed number of arms. We are also provided with context which in this particular case is context about user/users and news articles. Arms are analogous to news articles.

Contextual bandits algorithms pick an arm from the arm pool based on some algorithm. More specifically, the arm at time 't' is picked based on the history/parameters until time 't-1' and the prediction is then used to improve on the parameters corresponding the predicted arm.

There are several algorithmic variations of using contextual bandits for the personalization problem but the one explored in this assignment is called LinUCB algorithm which has been slightly modified to cater to the given dataset. The LinUCB algorithm essentially tries to find an upper bound for the difference in the expected value of predicted arm and the most optimal arm at that point.

In the usual datasets for contextual bandits, one is provided with feedback/reward for all the arms and consequently for the predicted arm as well, but in this dataset, we don't know necessarily know the reward for the arm that was predicted by the algorithm.

Pseudo code for the LinUCB algorithm is as below:

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 1. Pseudo code for the LinUCB algorithm

2. Dataset and its representation

Dataset 1:

This dataset has 102 columns. The first column is an article value from 1...10. The second column is the real-world reward on it. It is a binary value 0/1, where 0 indicates that the article in column 1 was displayed to the user and he didn't click on it. The value 1 means that the article in column 1 was displayed to the user and he clicked on it. The next 100 columns is the 100-dimensional feature vector.

Dataset 2:

This dataset has 101 columns. The first column is the article number ranging from 1....10. It's interpretation is that this article in column 1 was shown to the user and he clicked on it. All other articles were shown to the user and they didn't click on it.

3. Implementation and Results

I have implemented a modification of the LinUCB algorithm for the purpose of handling the fact that in the dataset 1, we don't necessarily know the reward of the arm that's predicted by the algorithm at time step 't' after learning for 't-1' trials. For the sake of solving this problem, I am making the update to the parameters corresponding to the predicted arm only when the predicted arm is equal to the given arm in the dataset. This is because we only know the reward (0 or 1) for the arm given in the dataset. The snapshot for the LinUCB() modified is as below. The code is also available in the python notebook if the below image isn't legible enough:

```
def LinUCB_modified(data_file, alpha=0.0, plot_or_not=True):
    articles = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    T = [x+2 for x in range(9999)]
    m = 10 # number of articles
    d = 100 # number of features
    total_payoff = 0.0
    time = 1 # to plot time series
    ctrs = {} # dict of time vs cumulative take rates
    ctr_num = 0.0 # initialising the CTR numerator
    ctr_den = 1.0 # initialising the CTR denominator
    matches = 0.0 # to record the number of matches between predicted arm and actual arm
    first = True
    with open(data_file, 'r') as data_file:
        lines = data_file.readlines()
        for line in lines:
            features = line.split(' ')
            real_article_chosen = features[0]
            reward = features[1]
            features = features[2:102]
            if first:
                first = False
                A = np.zeros((m, d, d))
                for a in range(10):
                    A[a] = np.eye(d)
                b = np.zeros((m, d, 1))
            ratings = np.zeros(m)
            for a in range(10):
                A_inv = np.linalg.inv(A[a])
                theta_a = A_inv.dot(b[a])
                features = np.asarray(features).astype(int)
                features = np.reshape(features, (100, 1))
                # uncomment the below when you want alpha based on time (1/sqrt(T))
                #alpha = find_optimal_alpha_1(time)
                ratings[a] = np.matmul(theta_a.T, features) + alpha * np.sqrt(np.matmul(np.matmul(features.T, A_inv), features))
            chosen_article = articles[np.random.choice(np.flatnonzero(ratings == ratings.max()))]
            if int(chosen_article) == int(real_article_chosen):
                # print("match at:",time)
                matches += 1.0
                total_payoff += float(reward)
                A[int(real_article_chosen)-1] += features.dot(features.T)
                b[int(real_article_chosen)-1] = b[int(real_article_chosen)-1] + int(reward)* features
                ctr_num += int(reward)
                # to account for the 1.0 initialization of ctr_den instead of 0.0
                if ctr_den != 0.0:
                    ctr_den += 1
                ctrs[time] = ctr_num/(1.0*ctr_den)
            else:
                ctrs[time] = ctr_num/(1.0*ctr_den)
            time += 1
    if plot_or_not:
        plot_cumulative_take_rate(ctrs)
    total_payoff /= matches
    return (ctrs[10000], total_payoff)
```

Figure 2. Code for LinUCB algorithm modified for dataset 1

To calculate the CTR, I have used the same formula as suggested in the homework guideline. To make the results more robust, I am calculating the CTR for the same value of alpha thrice. This because, my code uses a random tie breaking in the case when there are multiple arms with the same maximum payoff. Because of this, the algorithm can pick different arms in different runs in the few initial calls.

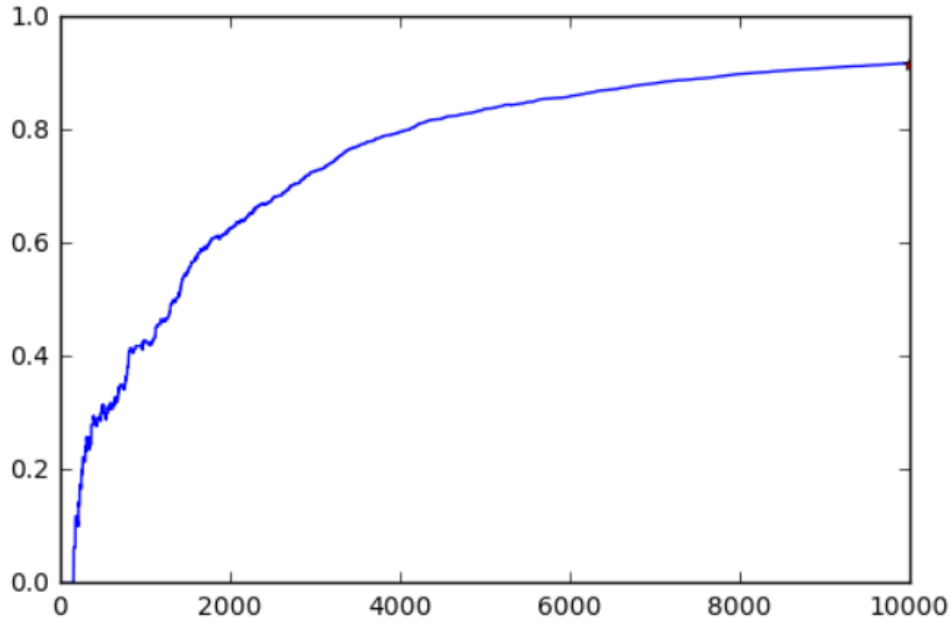
The three strategies to choose alpha are as follows:

1. One is the approach suggested in the homework guidelines. Alpha is equal to 1/sqrt(t)where t ranges from 2 to 10000. So instead of keeping alpha as a constant, it is dependent on time. The average CTR that I got for this type of alhpa is **0.916**. The code for this approach is as below:

```
def find_optimal_alpha_1(time):
    return 1/np.sqrt(time)
```

Figure 3. Code for time based alpha

The plot of time vs Cumulative take rate when alpha is chosen this way is shown below:




```
Cumulative Take rate: 0.9165103576099977, Total Payoff: 0.9173907848404492
CPU times: user 8min 10s, sys: 5min 29s, total: 13min 40s
Wall time: 1min 42s
```

Figure 4. Plot of time vs CTR

2. The second approach is the standard grid-search. After running single calls, I got an idea of the kind of range which will help reach a good alpha. The code for this approach is as below:

```
def find_optimal_alpha_2():
    values = np.arange(0.0001, 0.02, 0.001)
    ctrs = []
    for a in values:
        ctr, payoff = LinUCB_modified(DATA_FILE_1, alpha=a, plot_or_not=False)
        ctrs.append(ctr)
        print(str(a) + "---" + str(ctr))
    x = np.asarray(values)
    y = np.asarray(ctrs)
    plt.plot(x, y)
    max_index = np.argmax(y)
    max_ctr = y[max_index]
    max_alpha = x[max_index]
    print("alpha with maximum ctr: " + str(max_alpha))
    plt.plot(max_alpha, max_ctr, 'r*')
    plt.show()
```

Figure 5. Code for grid-search on alpha

Below is the plot of alpha vs CTR to find the optimal value of alpha which is giving the best CTR:

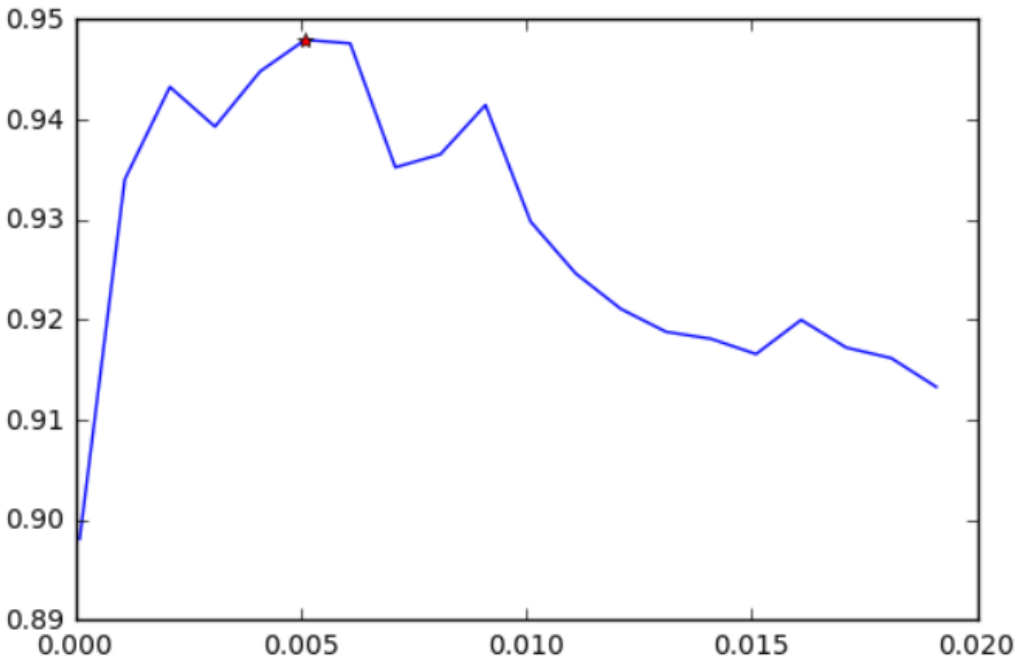


Figure 6. Plot of alpha vs CTR

The value of the optimal alpha as per the above plot is 0.0051 and the corresponding CTR is 0.948

3. In the third strategy, I am using the idea presented in one of the papers that uses a mathematical formula involving delta parameter which arises from trying to find an upper bound on the difference between the actual and expected reward.

$$\alpha = 1 + \sqrt{ln(2/\delta)/2}$$

From the formula, it seemed that alpha will definitely be greater than 1 and delta can't be above 2 as that will make the term under square root as negative which isn't possible. Hence that gives an idea about the bounds. I ran one iteration of the code with alpha = 1 and didn't receive a good value of CTR but this strategy is worth looking at because of its theoretical basis. The code for this strategy is as below:

```
def find_optimal_alpha_3():
    deltas = np.arange(1, 2, 0.1)
    ctrs = []
    for d in deltas:
        alpha = 1.0 + np.sqrt(math.log(2.0/d)/2)
        ctr, payoff = LinUCB_modified(DATA_FILE_1, alpha, False)
        ctrs.append(ctr)
        print(str(alpha) + "---" + str(ctr))
    x = np.asarray(deltas)
    y = np.asarray(ctrs)
    plt.plot(x, y)
    plt.show()
```

Figure 7. Code for delta based alpha

The plot of alpha vs CTR while choosing different values of delta is as below. The value of CTR is quite low as compared to above 2 options, and hence this strategy isn't as good but is definitely worthwhile to look at it:

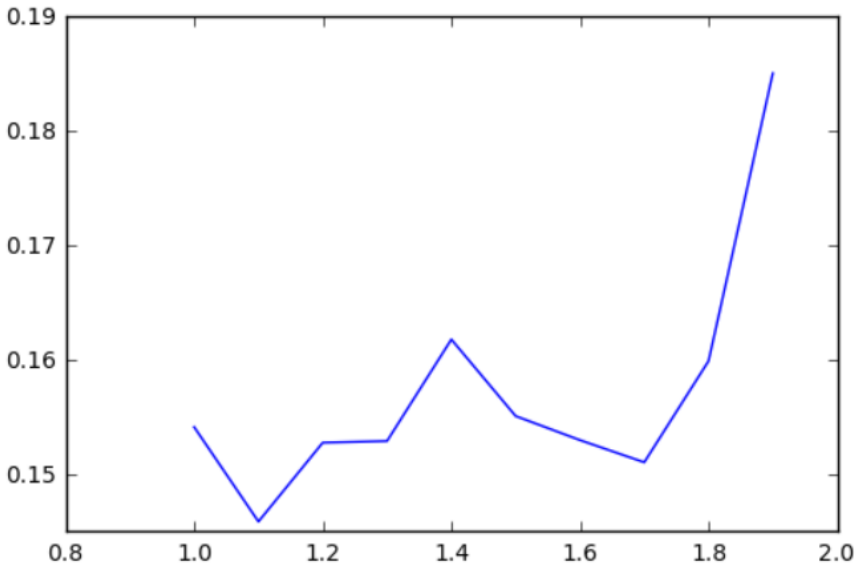


Figure 8. Plot of alpha vs CTR

I also implemented the **original LinUCB algorithm and ran it on the dataset 2.** This was to gain guarantee and confidence on my algorithm used on the dataset 1. I outputted the number of matches i.e. the number of times algorithm predicted the same article as the one that was clicked by the user. This number came out to be **9923 out of 10000** which is quite good. Below is the code for the same:

```
def LinUCB(data_file, alpha=0.0, plot_or_not=True):
    articles = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    T = [x+2 for x in range(9999)]
    m = 10 # number of articles
    d = 100 # number of features
    time = 1 # to plot time series
    matches = 0.0 # to record the number of matches between predicted arm and actual arm
    first = True
    with open(data_file, 'r') as data_file:
        lines = data_file.readlines()
        for line in lines:
            features = line.split(' ')
            real_article_chosen = features[0]
            features = features[1:101]
            if first:
                first = False
                A = np.zeros((m, d, d))
                for a in range(10):
                    A[a] = np.eye(d)
                b = np.zeros((m, d, 1))
            ratings = np.zeros(m)
            for a in range(10):
                A_inv = np.linalg.inv(A[a])
                theta_a = A_inv.dot(b[a])
                features = np.asarray(features).astype(int)
                features = np.reshape(features, (100, 1))
                # uncomment the below when you want alpha based on time (1/sqrt(T))
                #alpha = find_optimal_alpha_1(time)
                ratings[a] = np.matmul(theta_a.T, features) + alpha * np.sqrt(np.matmul(np.matmul(features.T, A_inv), features))
            chosen_article = articles[np.random.choice(np.flatnonzero(ratings == ratings.max()))]
            if int(chosen_article) == int(real_article_chosen):
                #print("match at: " + str(time))
                reward = 1.0
                matches += 1.0
            else:
                reward = 0.0
            A[int(chosen_article)-1] += features.dot(features.T)
            b[int(chosen_article)-1] = b[int(chosen_article)-1] + int(reward)* features
            time += 1
    return (matches)
```

Figure 9. Code for LinUCB algorithm for dataset 2

4. Learnings and Conclusion

This assignment was a great learning experience from the point of view of learning a new class of algorithms i.e. reinforcement learning. Contextual Bandits algorithm was implemented on the news article personalisation dataset and results were obtained in terms of cumulative take rates. Cumulative take rates essentially captured the number of times algorithm outputted the article which in reality when showed to the user was clicked by her. LinUCB algorithm was implemented for this purpose. This algorithm had to be slightly modified to manage the fact that dataset 1 didn't necessarily provided us with the reward (Click through value) of the article that was predicted by the algorithm.

Alpha which is a hyperparameter for the LinUCB algorithm controls the confidence bound. This value was estimated using 3-fold strategy as outlined in the report above. Plots of time vs CTR were plotted to see the progression of CTR with time which was overall increasing in nature.

5. References

1. Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms: Lihong Li, Wei Chu, John Langford, Xuanhui Wang, Yahoo! labs, 2012
2. A Contextual-Bandit Approach to Personalized News Article Recommendation: Lihong Li, Wei Chu, John Langford, Robert E. Schapire, 2010
3.http://banditalgs.com/
4.https://github.com/umeshksingla/news-recommend-ire