Replicating *A Gang of Bandits*

Bryce Bern BERNB@CARLETON.EDU
Carleton College, 300 North College St. Northfield Minnesota 55057 USA

Dawson d'Almeida DALMEIDAD@CARLETON.EDU
Carleton College, 300 North College St. Northfield Minnesota 55057 USA

Will Knospe KNOSPEW2@CARLETON.EDU
Carleton College, 300 North College St. Northfield Minnesota 55057 USA

Paul Reich REICHP@CARLETON.EDU
Carleton College, 300 North College St. Northfield Minnesota 55057 USA

Abstract

Recommendation systems are vital in maximizing user enjoyment. There exists an abundance of content for users to choose from on the many entertainment and service platforms available. Some platforms don't have access to preexisting information about what users would like. A next question might be how good can a recommendation be when we start with next to no information on our users as opposed to services used by companies such as Amazon or Netflix which have large swaths of data about every user. We can formalize this question about recommendation systems to be a multi-armed bandit problem. Recent research has sought to improve multi-armed bandits through better utilization of available information. *A Gang of Bandits* (Cesa-Bianchi et al., 2013) presents a novel algorithm that uses social network information and contextual information to more quickly learn users preferences and optimize cumulative payoff. This paper explores the process of replicating the experiments and results from *A Gang of Bandits* in order to validate the findings of the paper.

1. Introduction

1.1. Paper Replication

Paper replication is vital in all fields of research, as it tests both replicability of experiments and validity of results. In order for previous research and findings to act as strong, trustworthy bases for further research, their methodology and results must be replicable. This paper describes the process of replicating *A Gang of Bandits*, a machine learning research paper that presents a novel recommendation system.

1.2. Recommendation Systems and Multi-Armed Bandits

In an era where an uncountable number of online services are available to users, it's vital for these services to provide satisfying recommendations for the users. As popularity is volatile and available content is constantly changing, recommendation systems must try to learn users' interests in order to properly map them to available content.

Multi-armed bandit problems formalize this trade-off of attempting to learn more about users' preferences versus showing users content that we're confident they would like. The conventional definition of the multi-armed bandit problem is where a person has k indistinguishable choices, each of which has a probabilistic reward. Contextual multi-armed bandits, like those that are discussed in this paper, are slightly different in that each choice has a context that might hint at which arm will give a higher reward. (Chu et al., 2011) Because online advertisements or recommendation systems have information about their users, contextual bandits are a more suitable subject of study than non-contextual bandits. However, recommendation systems also use social networks like Facebook to connect friends. And while there has been some research of non-contextual multi-armed bandits within a social network (Buccapatnam et al., 2013), there has not been any study of contextual multi-armed bandits

Algorithm 1 Problem Space

```
for t in 1...T do
    Select user i, contexts {x_1, ..., x_t} from dataset
    x = learner.choose(user i, contexts {x_1, ..., x_t})
    Calculate payoff a for context x with user i
    learner.update(context x, payoff a)
end for
```

in a social network previous to *A Gang of Bandits*.

2. Formalizing the Problem Space

Before developing the details of our algorithms, it would be useful to clearly define our terminology and problem space. An instance of one of our algorithms is a *learner*, which has associated actions choose and update. The items that we choose between (which could be songs, websites, TV shows, etc.) are *contexts*, and in the choose action the learner chooses a context to recommend. Each context has an associated *context vector*, which represents each context, and is used to provide additional information during the choose action. For context vectors x, y, if x and y represent "similar" contexts, then we expect the dot product xy^T to be relatively high. Should x and y represent "different" contexts, then we expect the dot product to be relatively low. In the case of our real-world datasets, x and y are generated using user-defined tags from the dataset. Finally, the person who we are selecting contexts for is the *user*. Depending on whether the user likes a recommended context, the learner updates its knowledge and learns from the user's payoff in the update action. With this in place, our algorithms then operate in the problem space outlined in Algorithm 1.

When performing the choose and update actions, our algorithms can either rely solely on the context information encoded in the context vectors, or incorporate the information regarding the social network of users as well, represented as a graph and presumed to be included in the construction of the learner.

3. Algorithms

We now move on to the algorithms presented in *A Gang of Bandits*. Two primary algorithms are presented, LinUCB, a contextual bandit algorithm that relies only on context information, and GOB.Lin, which is an original algorithm presented in *A Gang of Bandits* that relies additionally on the social network graph information. In addition, two variants of GOB.Lin operating on clustered user graphs are presented, namely GOB.Lin.BLOCK and GOB.Lin.MACRO, which use separate GOB.Lin instances for each cluster and a single GOB.Lin instance across all clusters. The first of these algorithms we present is LinUCB, which provides both a baseline for the results presented in the paper and the underlying framework for the original algorithm presented in *A Gang of Bandits*, GOB.Lin.

3.1. LinUCB

LinUCB is an algorithm for a contextual linear bandit that incorporates contextual information but not social network information. In any contextual linear bandit, we are assuming that payoffs are determined as a linear function of the context vectors: that is, there exists some vector u such that the payoff a of a context vector x_i is the dot product:

$$ux_i^T$$

Essentially, our algorithm is attempting to do its best to estimate u by observing the payoffs of each context. Our algorithm will maintain a vector w that estimates u, and update it to try and improve its estimate of u as we observe more payoffs.

Before examining exactly how this update will occur, we will discuss how the choose action of the LinUCB learner occurs. The simplest method of choosing a context would be to choose the context with the highest estimated score: the context with the maximum wx_i^T. If w is already a good estimation of x_i, then this is effective, but otherwise it may be useful to "explore" more different context vectors in order to get more information about w. We represent the desire to explore using the confidence bound CB, which is high when we don't know much about a given context and low when we've already seen that context many times. As a result, we will choose the context with the maximum value for:

$$wx_i + CB(x_i)$$

In order to maintain information about both the approximation vector w and which vectors we have seen for the confidence bound CB, our algorithm will maintain a matrix M, which stores information about which context vectors we have seen, and a bias vector b, which stores information about the kind of context vectors produce good payoffs. The pseudocode of LinUCB can then be found in Algorithm 2. *A Gang of Bandits* modified $\sqrt{ln(|M_t|/\delta) + ||u||)}$ in Algorithm 2 to an approximation relying on the timestep t coupled with an exploration rate α in their implementation, as this produces considerable speed increases with no discernable decrease in performance. δ is a constant and is folded into our new α value.

Algorithm 2 LinUCB

Init: $b_0 = 0 \in R^d$ **and** $M_0 = I \in R^{d \times d}$
for t in 1...T **do**
  Set $w_{t-1} = M_{t-1}^{-1} b_{t-1}$
  Select user i, contexts $\{x_1, ..., x_t\}$ from dataset
  Set:
  $k_t = argmax_{k=1,...,n}(w_{t-1}^T x_{t,k} + CB_t(x_{t,k})$
  Where:
  $CB_t(x_{t,k}) = \sqrt{x_{t,k}^T M_{t-1}^{-1} x_{t,k}}(\sigma\sqrt{ln(\frac{|M_t|}{\delta})} + ||u||)$
  Calculate payoff a for context x with user i
  Set $\bar{x_t} = x_{t,k_t}$
  Observe reward $a_t \in [-1, 1]$
  Update:
  $M_t = M_{t-1} + x_t x_t^T$
  $b_t = b_{t-1} + a_t x_t$
end for

We now will examine a specific case of LinUCB for intuition as to its function. Consider the case at which point we have observed a single vector x_1 with payoff a_1. Then, currently, our matrix M_t is:

$$I + x_1 x_1^T$$

And b_t is $a_1 x_1$ Now consider observing a new context vector x. Our score is:

$$(I + x_1 x_1^T)^{-1}(a_1 x_1)x + CB(t)$$

The Sherman-Morrison formula says that:

$$(A = uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1} u}$$

We can use this to simplify the inversion to:

$$(I - \frac{x_1 x_1^T}{1 + x_1^T x_1})(a_1 x_1)x + CB(t)$$

Multiplying this out, we get:

$$a_1 x_1 x - \frac{(x_1 x_1^T)(a_1 x_1)x}{1 + x_1^T x_1} + CB(t)$$

Clearly, $a_1 x_1 x$ is maximized when x is most similar to x_1. We can use the associative property and factor out a_1 to get:

$$a_1(x_1 x - \frac{x_1(x_1^T x_1)x}{1 + x_1^T x_1}$$

$(x_1^T x_1)$ is a scalar, so we can factor to:

$$a_1 x_1 x(1 - (x_1^T x_1)/(1 + x_1^T x_1))$$

Writing 1 as $(1 + x_1^T x_1)/(1 + x_1^T x_1)$:

$$a_1 x_1 x(\frac{1 + x_1^T x_1 - x_1^T x_1}{1 + x_1^T x_1}) = \frac{a_1 x_1 x}{1 + x_1^T x_1}$$

The numerator is easy to understand: the more similar x is to x_1, the higher score we will give if a_1 is positive, and the lower the score if a_1 is negative. The denominator essentially provides a weight representing the number of times x_1 has been sampled. As the number of terms in this sum increases, the denominator will weight each score appropriately. The confidence bound can be understood similarly, as it depends primarily on:

$$x^T(I + x_1 x_1^T)^{-1}x = x^T x - \frac{(xx_1)(x_1^T x))}{(1 + x_1^T x_1)}$$

Which we can see is maximized when x and x_1 are not similar and minimized when the are, at which point $\frac{(x^T x_1)(x_1^T x)}{(1+x_1^T x_1)}$ scales to be much like $x^T x$. The confidence bound is maximized when vectors are dissimilar to any vector we have seen before because those vectors will provide the most new information. Note that in the above algorithm we are agnostic to any features of the user. LinUCB has two variants for dealing with multiple users: firstly, LinUCB-IND runs a separate instance of LinUCB for each user, allowing each instance to specialize to a user's taste, whereas LinUCB-SIN runs a single instance of LinUCB for all users, which learns faster and performs well if all users are relatively similar.

3.2. GOB.Lin

While LinUCB does well at estimating payoffs, we might ask ourselves whether any additional information derived from a social network graph can be incorporated into our learning and decision-making process. For GOB.Lin, we assume that each user i has its own linear function for payoffs, which we will call u_i. In addition, we assume that if users i and j have an edge between them in the social network graph across all the users, then u_i and u_j will be relatively similar. As a result, it makes sense that when we observe a new context vector and payoff x_t and a_t with user i, we should update the other approximation vectors w_j in addition to w_i, scaling based on the closeness between j and i.

In order to implement this, we essentially arrange the matrices from LinUCB into blocks along the diagonal of a larger matrix M of size $dm \times dm$, where d is the length of a context vector and m is the number of users. Likewise, we arrange the bias vectors into a larger vector b of size dm. Then, when observing a context vector x with a particular user i, we first shift x into a new vector ϕ padded with zeros, such that x aligns with the ith block of M. As described, performing the algorithm from LinUCB with these shifted

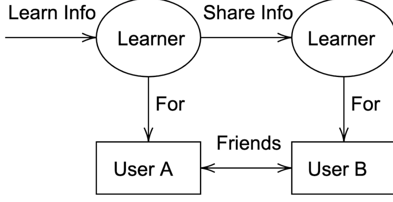

Figure 1. How We Update Associated Learners

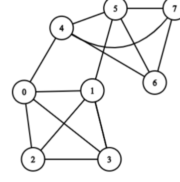

Figure 2. A base user graph

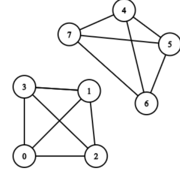

Figure 3. The user graph modified for GOB.Lin.BLOCK

context vectors and modified matrix and bias would be equivalent to LinUCB-IND, as the padded zeros would prevent any matrix or section of the bias vector other than the selected one for user i from contributing to our calculations. As a result, in order to incorporate our graph information, we use a modified representation of our graph to "spread" the context information across connected blocks. This modified representation is the matrix $A_{\otimes}^{-\frac{1}{2}}$, where $A_{\otimes}$ is the kronecker product of a matrix A and I_d, and where A is $I_n + L$, where L is the Laplacian of our graph.

Algorithm 3 GOB.Lin

Init: $b_0 = 0 \in R^{dn}$ **and** $M_0 = I \in R^{dn \times dn}$
for t in 1...T **do**
Get $i_t \in V$, context $C_{i_t} = \{x_{t,1}, ..., x_{t,c_t}\}$;
Construct vectors $\phi_{i_t}(x_{t,1}), ..., \phi_{i_t}(x_{t,c_t})$, and modified vectors $\tilde{\phi}_{t,1}, ..., \tilde{\phi}_{t,c_t}$, where

$$\tilde{\phi}_{t,k} = A_{\otimes}^{-\frac{1}{2}} \phi_{i_t}(x_{t,k}),\ k = 1, ..., c_t;$$

Set:

$$k_t = argmax_{k=1,...,c_t}(w_{t-1}^T x_{t,k} + CB_t(x_{t,k}))$$

where:

$$CB_t(\tilde{\phi}_{t,k_t}) = \sqrt{\tilde{\phi}_{t,k_t}^T M_{t-1}^{-1} \tilde{\phi}_{t,k_t}} \left(\sigma \sqrt{ln(\frac{|M_t|}{\delta}) + ||\tilde{U}||}\right)$$

Observe reward $a_t \in [-1, 1]$ with user i_t
Update:
$M_t = M_{t-1} + \tilde{\phi}_{t,k_t} \tilde{\phi}_{t,k_t}^T$
$b_t = b_{t-1} + a_t \tilde{\phi}_{t,k_t}$
end for

Note that the size of the matrices we are working with is greatly increased in comparison to LinUCB. We will introduce clustered algorithms to improve this.

3.3. Clustered Algorithms

M now scales in size with the number of users. M must be inverted every timestep, and while this can be done using the Sherman-Morrison formula to perform this inversion in $O(n^2)$, this still leads to the algorithm's runtime scaling quadratically. As a result, *A Gang of Bandits* introduces the use of clustering of users on the social network graph to reduce the number of users being considered at once.

The clustering of the users in *A Gang of Bandits* was performed using the Graclus algorithm, an implementation of which is available online. There were two primary methods presented for using clustering, namely GOB.Lin.BLOCK and GOB.Lin.MACRO.

3.3.1. GOB.Lin.BLOCK

GOB.Lin.BLOCK is implemented by running separate instances of GOB.Lin on each cluster in our social network graph, maintaining edges within the cluster but isolating it and cutting it off from the rest of the graph. Each disconnected graph has a small number of users, and so our runtime is decreased as a result. Intuitively, we expect that GOB.Lin.BLOCK, with its ability to specialize each learner instance to specific user clusters, will perform well where users have diverse preferences. As a special case, if all users are clustered separately, then this is equivalent to running LinUCB-IND. We can see a visualization of the clusters of GOB.Lin.BLOCK in 3.

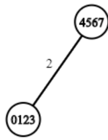

Figure 4. The user graph modified for GOB.Lin.MACRO

3.3.2. GOB.Lin.MACRO

GOB.Lin.MACRO is implemented by running one instance of GOB.Lin and treating each cluster in our social network graph as a single user, with the edges between these new "users" weighted by the number of inter-cluster edges in the original graph. Intuitively, we expect that GOB.Lin.MACRO will do best when users have similar preferences, as we have one learner that is worse at specializing. In a corresponding special case, if all users are clustered together, then this is equivalent to running LinUCB-SIN. We can see a visualization of the clusters of GOB.Lin.MACRO in 4.

4. Datasets

4.1. Overview

The researchers tested the GOB.Lin algorithm against LinUCB on an artificial dataset called 4Cliques and two real publicly available datasets from the social bookmarking web service Delicious and from the music streaming service Last.fm.

Although the researchers were not explicit about why they decided to construct 4Cliques, we believe it was used to test GOB.Lin's ability to take advantage of a social network. In 4Cliques,a user's preferences are directly related to all other users' within their clique, creating a perfect environment to test if Gob.Lin is correctly exploiting social information.

On the other hand, the Delicious and Last.fm are instances where we would expect social information to matter, but don't know for certain. So while they are excellent ways to measure the performance of Gob.Lin as compared to Lin.UCB, they would not be useful in trying to see if Gob.Lin is correctly using social information.

4.2. Creating 4Cliques

The graph of 4Cliques starts as 4 cliques of 25 nodes each, with a user u_i having the same randomly generated 25-dimensional unit preference vector as all other users in its clique. All initializations of 4Cliques also take 2 parameters as input, graph noise and payoff noise. We then generate a 100 x 100 symmetric matrix of random numbers between [0,1]. All elements of the matrix above the inputted value for graph noise is set to 1 and all others are set to 0. This noise matrix is then XORed with the original graph matrix to obtain a noisy version of the graph.

Because 4Cliques is an artificial dataset, we have to create contexts for the algorithm to compare. At every timestep, we generate a set of 10 random randomly generated 25-dimensional unit context vectors. Payoffs for each context vector x and user are calculated with the following equation:

$$u_i^T x + \epsilon$$

The dot product between u_i^T and x just compares how similar a selected user's preferences are to the chosen context, and ϵ is randomly generated from an interval centered at 0 and bounded by the payoff noise parameter.

4.3. Last.fm and Delicious

The choice of real world datasets was due to the different structure of user preferences as they relate to their friends' preferences. Specifically, user preferences in Delicious are more varied than they are in Last.fm. This is because there are more popular artists, the contexts in Last.fm, that everyone enjoys than there are popular websites, the contexts in Delicious. The difference between the two datasets gives a more accurate reflection of the performance of recommendation algorithms.

Figure 5 gives some of the main statistics for both datasets, where items count the number of artists in Last.fm and the number of URLs in Delicious, and Non-Zero payoffs is the number of user-item pairs that have a non-zero payoff.

5. Implementation

Our preprocessing of the dataset followed all the steps outlined in *A Gang of Bandits*. We began by creating a social network adjacency matrix, where a 1 in the matrix represented an edge between the column and row users. We then processed tags in the given datasets. For Last.fm, we split the artists' ids and the tag ids associated with a user's tag. We then created a dictionary that stored a list of tags associated with any given artist. We also split all tag names containing underscores and hyphens, as tags like "rock_and_roll" and "rock-n-roll" could both apply to one artist. This was specifically mentioned by the paper, as it decreased

	Last.fm	Delicious
Nodes	1892	1867
Edges	12717	7668
Avg. degree	13.443	8.21
Items	17632	69226
Nonzero payoffs	92834	104799
Tags	11946	53388

Figure 5. Statistics for Delicious and Last.fm

the number of unique tags significantly. For Delicious, we processed the tags very similarly, but every website was directly associated with its tags. We again split tag names, but we also removed all tags that appeared less than ten times in order to reduce the size of the tag set.

In order for LinUCB and GOB.Lin to use the datasets, we had to provide them with context vectors. In order to do this for Delicious and Last.fm, we created TF-IDF context vectors to uniquely represent contexts, which places higher emphasis on more unique tags. We used sklearn's base TfidfTransformer without altering any parameters and applied it to a simple matrix that associated contexts with tags. We then compressed our high-dimensional sparse representation of each context into a 25 dimensional dense representation. This retained only the first 25 principle components of each context vector, as described in the paper, using sklearn's TruncatedSVD to do so. 4Cliques' contexts were uniformly random unit vectors of size 25.

At each time step, we generated 25 random contexts for Delicious and Last.fm or 10 random contexts for 4Cliques. For Delicious and Last.fm, we guaranteed that at least one of the 25 vectors would result in a payoff of 1, meaning the user had bookmarked the site (Delicious) or listened to the artist (Last.fm). This was specified in the paper and it removed meaningless comparisons.

Additionally, when adding noise to the 4Cliques graph, we only applied the noise to the top triangle of the adjacency matrix. We then copied it to the bottom half. This was done in order to maintain symmetry of the adjacency matrix.

An important thing to note is that on every test run of any given algorithm using either the Delicious or Last.fm datasets, the TF-IDF and SVD were used to generate context vectors for all contexts. As sklearn's TruncatedSVD is non-deterministic, every test run can result in different context vectors.

When implementing both LinUCB and GOB.Lin, we used numpy arrays and matrices. In order to construct our laplacian matrix, we used sp_sparse's laplacian function on our network graph. Additionally, all doubles within these matrices were represented as float32's. In the implementation of LinUCB, we treated LinUCB-SIN as LinUCB-IND run for only one user. We also used the Sherman Morrison formula to decrease the runtime of matrix inversions from n^3 to n^2. This inversion is done on the context matrix, which is the sum of an invertible matrix and an outer product.

In order to implement the two cluster variants of GOB.Lin, we had to cluster the network graphs of both Delicious and Last.fm. We used graclus, a graph clustering software written in C that was specified by the paper, to do this. We interpreted the paper's cluster sizes to be the number of clusters to create, as the software has no way of specifying the number of nodes in a cluster. We ran the clustering software once for each specified number of clusters (5, 10, 20, 50, 100, 200) and stored the node to cluster associations in csv's for later use.

Our implementation of GOB.Lin.MACRO ran a single instance of GOB.Lin on the clustered graph. For any given time step, the given user's cluster was found, and that super-node and all connected super-nodes were updated as if they were single users. This signified updating all individual users in those clusters. As such, a single context matrix and bias vector were sufficient for any iteration of GOB.Lin.MACRO.

For GOB.Lin.BLOCK, we instead ran many instances of GOB.Lin on the different clusters. Given a user, we would find its cluster and use the context matrix and bias vector associated with that cluster, updating only those values. As such, there were an equal number of context matrices and bias vectors to clusters for any iteration of GOB.Lin.BLOCK.

6. Results

In this section, we examine the results generated when running each algorithm on the datasets mentioned earlier: 4Cliques, Last.fm, and Delicious. We are using LinUCB as a baseline point of comparison to determine how well each version of GOB.Lin is performing. Additionally, we normalize each of the algorithm's cumulative reward against an agent choosing contexts randomly. As we examine the results obtained using our implemented algorithms, we will compare our findings to that of the researchers'. We will begin by inspecting the cumulative reward graphs for 4-Cliques.

6.1. 4-Cliques

In figure 6, we can see that our findings for 4Cliques determine that GOB.Lin performs the best, reaching

a cumulative reward of around 3,000, while LinUCB-IND and LinUCB-SIN trail, respectively reaching cumulative rewards of around 2,600 and 1,600. These results resemble the researchers' findings very closely. We also ran the algorithms on 4Cliques with different values for the graph noise and payoff noise parameters. In these cases, our graphs for cumulative reward once again matched the researchers'.

6.2. Delicious

The first thing to note when inspecting figure 7 is the lack of GOB.Lin. Unfortunately, due to the quadratic scaling of the matrix inversion, running GOB.Lin on datasets as large as Delicious and Last.fm was prohibitively slow. As for the rest of the multi-armed bandit algorithms, we discover GOB.Lin.BLOCK and GOB.Lin.MACRO outperforming both versions of LinUCB. GOB.Lin.MACRO does the best overall, and LinUCB-IND does slightly better than LinUCB-SIN. For the Delicious dataset, we know that users tend towards dissimilar interests. And so, it makes sense that GOB.Lin.BLOCK does better than GOB.Lin.MACRO since Block stores a matrix for every individual users' preferences, whereas Macro clumps users together, treating them all as one. Under the same reasoning, LinUCB-IND does better than LinUCB-SIN; LinUCB-IND is able to store personalized data for each user, while LinUCB-SIN assumes that users are all going to have similar preferences. When comparing our results to the researchers', we see that both follow the same trends, but the scale between the graphs is different. While GOB.Lin.BLOCK does the best in both cases, our version achieves a cumulative reward of 600, while their version only reaches around 125. The rest of the algorithms follow this same scaling trend. We will talk more about why we think that this discrepancy exists in the conclusion section.

6.3. Last.fm

Just as with Delicious, we were unable to obtain the results for GOB.Lin due to the size of Last.fm. In figure 8, LinUCB-SIN performed the best, with GOB.Lin.MACRO taking a close second. Both GOB.Lin.BLOCK and LinUCB-SIN have a negative cumulative reward, meaning that they are choosing contexts worse than a random agent would. These finding also follow the trends we would expect for the Last.fm dataset; Last.fm contains users who all have very similar preferences. And so, it makes sense that LinUCB-SIN does the best, since it treats every user as having the same preferences, and GOB.Lin.MACRO does almost as well, since it treats each cluster as a single user. While our findings follow similar trends to the researchers', they do not match entirely. The researchers' graph shows that GOB.Lin.MACRO does the best, with LinUCB-SIN in a close second, but our findings switch the two. How well or poorly our algorithms perform is also magnified in comparison to the researchers'. Our results show that LinUCB-SIN and GOB.Lin.MACRO both reach around 3,000, whereas the researchers calculate a reward closer to 1,250. Our results are also extreme in the other direction; LinUCB-IND and GOB.Lin.BLOCK collect a negative reward compared to the researchers' findings, where Block and LinUCB-IND have rewards of 750 and 175 respectively.

7. Conclusion

When running the algorithms we implemented on 4-Cliques, our findings closely resemble the researchers'. We consider this a success as it acts to validate the researchers' work and bolsters the foundation for future research to be conducted in the future. For both Last.fm and Delicious, our results follow the same general trends as the researchers' but differ regarding scale. We are confident that our algorithms are implemented correctly because we are producing the same results as the researchers on the 4-Cliques dataset. Therefore, we believe that this discrepancy is due to how the two datasets are being processed. In the researchers' explanation of processing the data, they are not clear in exactly what functions and parameters they use to implement TF-IDF and PCA. If the difference in dataset processing is causing this discrepancy, it may imply that these algorithms are not as robust to changes in the dataset as they might appear by reading the paper. In the future, further experimentation into how different dataset processing impacts the cumulative reward for these algorithms would be valuable.

References

Buccapatnam, Swapna, Eryilmaz, Atilla, and Shroff, Ness B. Multi-armed bandits in the presence of side observations in social networks. In *52nd IEEE Conference on Decision and Control*, pp. 7309–7314. IEEE, 2013.

Cesa-Bianchi, Nicolo, Gentile, Claudio, and Zappella, Giovanni. A gang of bandits. In *Advances in Neural Information Processing Systems*, pp. 737–745, 2013.

Chu, Wei, Li, Lihong, Reyzin, Lev, and Schapire, Robert. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 208–214, 2011.

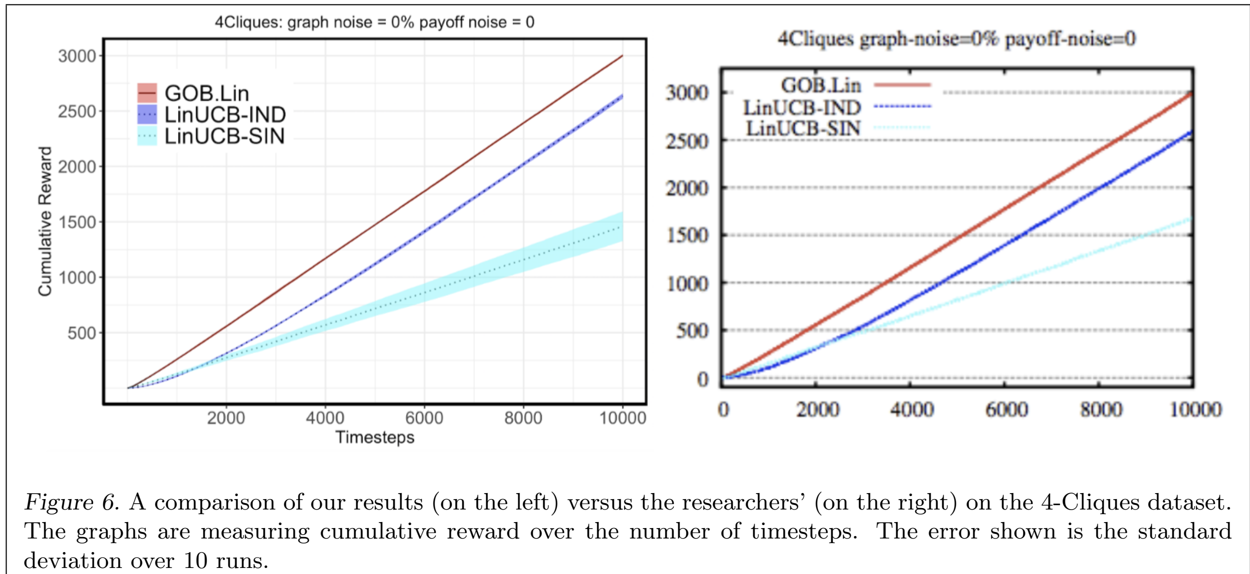

Figure 6. A comparison of our results (on the left) versus the researchers' (on the right) on the 4-Cliques dataset. The graphs are measuring cumulative reward over the number of timesteps. The error shown is the standard deviation over 10 runs.

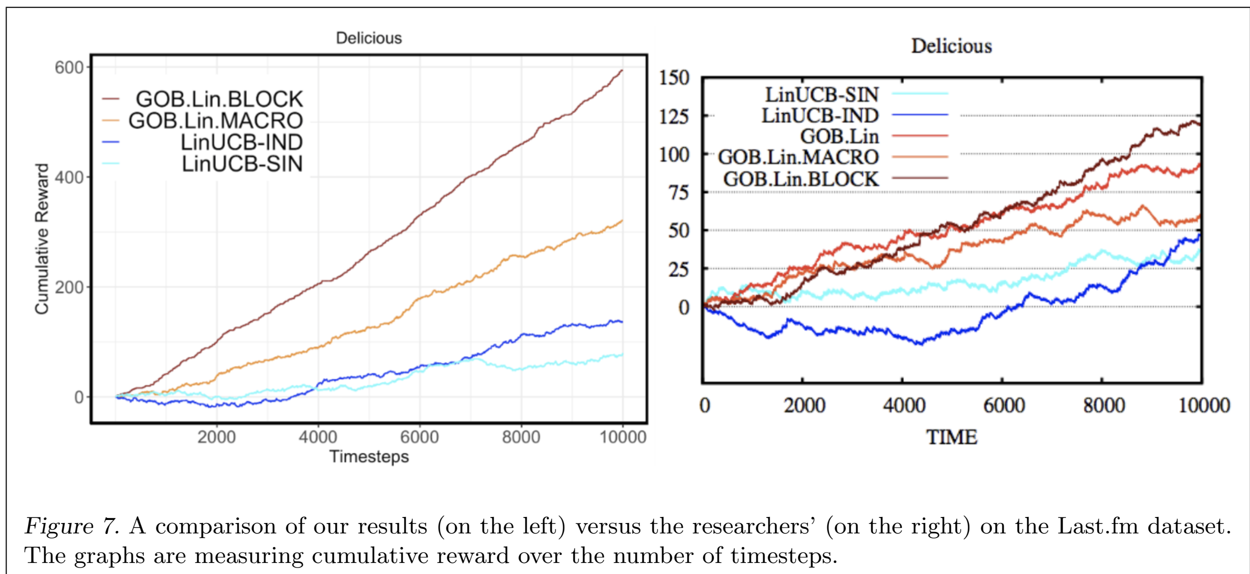

Figure 7. A comparison of our results (on the left) versus the researchers' (on the right) on the Last.fm dataset. The graphs are measuring cumulative reward over the number of timesteps.

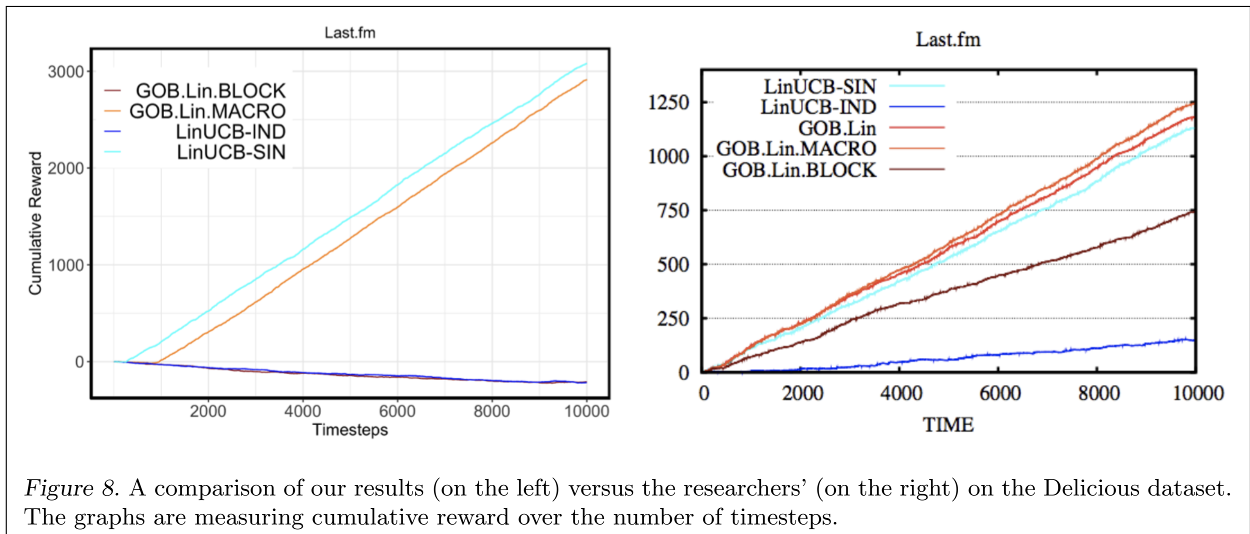

Figure 8. A comparison of our results (on the left) versus the researchers' (on the right) on the Delicious dataset. The graphs are measuring cumulative reward over the number of timesteps.