# Contextual Bandit Approaches to Personalized Tutoring

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Alan Healy

Project Advisors:

Dr. Neil Heffernan
Ethan Prihar

Sponsored By:

ASSISTments

Date: March 2022

## Abstract

In order to choose an algorithm to recommend student tutoring in the ASSISTments platform, a simulator was created based on historical ASSISTments data. Various contextual bandit models were tested in that simulator. It was found that Disjoint LinUCB and Hybrid LinUCB obtained the highest cumulative reward over a period of 100,000 trials. Consequently, Disjoint and Hybrid LinUCB will be implemented in the ASSISTments Reinforcement Learning Service to improve student learning outcomes.

# Contents

# 1 Introduction

The ASSISTments learning platform contains a robust system of tutoring for students, in which many mathematics problems have multiple options for tutoring, such as videos and written hints. Not all students will respond well to every type of tutoring, so it is a goal of ASSISTments researchers to find the best ways to choose from available tutoring strategies. This project investigates the effectiveness of multiple algorithms for choosing student tutoring based on prior ASSISTments data.

## 1.1 Project Objectives

The goal of this project is to choose an algorithm that will help students learn on the ASSISTments platform by simulating tutoring. We can achieve this by first building a simulator for student tutoring, populated with data gathered in ASSISTments. We can then create models to choose which tutoring to give a student, and test how well that student performs given that tutoring.

### 1.1.1 Build a Simulator for Student Tutoring

Our first objective is to build a simulator which will evaluate various decision-making models on their ability to choose the best tutoring for students. This simulator will feed data about a particular problem and a particular student into the algorithm, as well as provide a list of choices for tutoring (such as hints or videos) with associated data. The model will choose which tutoring to provide the student, and the simulator will evaluate whether that tutor strategy helped the given student answer the next problem correctly. The simulator will then store the results of this evaluation in a database. Repeating this evaluation thousands of times, we can quantify how well the model performs based on the total number of times the student answered the next problem correctly.

### 1.1.2 Create Contextual Bandit Models to Run in the Simulator

In educational experiments, different types of tutoring are given to students to observe their effects. However, if a student is given an experimental treatment which turns out to be ineffective, it can be problematic for the student's long term learning. To mitigate this, we want to use an algorithm that will respond to a low performing tutoring strategy by altering the experimental design such that future students receive the ineffective treatments less.[1] Multi-armed bandit algorithms are particularly helpful in this scenario, as they balance the need to get useful statistical data with the need to not severely impact students' learning.[1]

In this project, we will specifically focus on contextual bandit algorithms, a class of multi-armed bandit algorithms which make use of contextual data regarding students, problems, and choices of tutoring.

Each model we build must contain the following.

- A "recommend" method, which takes in a set of features about a student, a problem, and a list of tutoring strategies, then outputs a choice of tutoring strategy.
- An "update" method, which takes in the choice of model made in the recommend method as well as the reward obtained from that recommendation. The update method changes the state of the model to reflect what it has learned from the result of the previous recommendation.

#### 1.1.3 Test the Performance of the Models

Since we're trying to recommend tutoring to students, we need to make sure our models perform well. Thus, we need a way to evaluate the performance of our models. After running the simulator for a model's recommend and update methods several thousand times, the total reward measures how well that model performed. In order to obtain a metric for evaluating the model's performance, we compare it to the total reward of a random model tested the same number of times.

## 2 Background

Before we discuss the simulator and models we built, we must first explain what ASSISTments is and why bandit models are useful for our project.

### 2.1 ASSISTments

ASSISTments is an online learning platform focusing on K-12 mathematics. In this ecosystem, every user is either a teacher, student, or researchers. Teachers can assign work to a class, students can see when they've been assigned work and can complete homework assignments online. While completing assignments, students work in the ASSISTments tutor. One of the main goals of ASSISTments is to identify when students are struggling with material, and provide the tools to help them learn. To that end, teachers will receive reports about how each student in their class is performing on each type of problem assigned. This enables teachers to give more focused tutoring when students struggle in particular topics.[2] ASSISTments is a large scale platform, and teachers may not have time to provide individual tutoring to every student who is struggling. To address this, ASSISTments started the TeacherASSIST program.

The TeacherASSIST program is a crowdsourced collection of tutoring for K-12 mathematics problems.[3] While completing a problem, students who are struggling may have the option to request support. This support may come in the form of a hint, explanation, video guide, or simply the answer of the problem. When multiple forms of support are available for a problem, TeacherASSIST may randomly assign a tutoring strategy to a student. Then, TeacherASSIST collects data about whether the student answered the next problem correctly after receiving tutoring. This helps inform researchers about what types of tutoring strategies may be helpful for particular users. Instead of randomly assigning tutor strategies, ASSISTments researchers want to assign tutor strategies based on an algorithm to provide the best tutoring for a student.[4] We can view the problem of assigning the optimal tutoring to a student as a multi-armed bandit problem.

## 2.2 Multi-Armed Bandit Problems

Multi-armed bandit problems are a class of problems in which an automated agent tries to acquire knowledge as well as exploit its current knowledge to make the best possible choices.[5] Algorithms used to address these problems are known as multi-armed bandit algorithms. The classic example used to illustrate multi-armed bandit problems is as follows. Consider a row of slot machines (or "one-armed bandits") which each have different expected payoff amounts. If the slot machines can be pulled some finite number of times, how can we obtain the most cumulative reward with our finite number of pulls? One strategy would be to pull each slot machine one time and observe their payoffs. Then, we pull whichever slot machine awarded the most reward on the initial pull for the rest of our pulls. This strategy is flawed if, for example, the slot machine which originally awarded the most payout actually has a lower expected value than another slot machine that happened to not award as much on the first pull. In this case, we say the strategy focuses too much on "exploitation" of existing knowledge, and not enough on "exploration" of new knowledge.[6] Another strategy would be to repeat the previous strategy, but occasionally pull a random slot machine and update the expected values of each slot machine every time we pull one. This approach balances exploration and exploitation more than the previously described strategy. Finding the right ratio of exploitative actions and exploratory actions is key to obtaining the most cumulative reward with a bandit model.[7]

Multi-armed bandit problems appear in many different fields, particularly in experiments where some number of outcomes have different expected payoffs, such as medical trials or resource allocation.[8] As a result, they have been widely studied and various approaches have been created to address them. One class of multi-armed bandit algorithms have been growing in popularity with the development of the Internet: contextual bandit algorithms.

### 2.3 Contextual Bandit Algorithms

We can add context information to bandit models to get a new class of algorithm called contextual bandit algorithms. These allow us to approach a wider variety of problems with better results than standard bandit algorithms. Contextual bandit algorithms are useful in personalized recommendations, such as for news articles[9] and educational tutoring.[1] Typically, a "context vector" will contain some number of features about a particular entity in the contextual bandit problem, such as a user or a news article. In the case of a user, a context vector may store information such as the typical time spent reading articles on a website per day, or statistics on how often the user clicks on recommended articles.[10] Contextual bandits are useful when we don't know which features are statistically significant in achieving a higher reward. For example, it may be the case that gender identity affects what type of news article a user prefers. The contextual bandit algorithm will take into account many of these features when recommending an article, and with sufficient training it can produce better recommendations.

## 3 Methodology

In order to create a high-performance contextual bandit for student tutoring recommendations, we chose a series of existing non-contextual and contextual bandit algorithms and created a simulator to evaluate their performance.

### 3.1 Algorithms Used

We used the following algorithms in this investigation.

- Non-contextual Random Model
- Non-contextual Thompson Sampling
- Pooled Linear Thompson Sampling
- Disjoint Linear Thompson Sampling
- Hybrid Linear Thompson Sampling

- Pooled LinUCB
- Disjoint LinUCB
- Hybrid LinUCB

Each of these algorithms takes in (but does not necessarily make use of) lists of features for a problem, a student, and various tutoring strategies. The algorithms first decide on which tutoring strategy to provide the student. Then, the reward for that decision is fed back into the algorithm so it can update its parameters. We chose to implement these models and the simulator in python for its variety of data processing software.

### 3.1.1 Random Model

The Random Model is the simplest model used in this investigation. It uses a random number generator to choose a tutoring strategy to provide the student. This model is used as a benchmark to compare other models. If another model consistently obtains more reward in a given number of trials than the random model, it may be useful to implement in ASSISTments. Likewise, if a model does not perform better than the random model consistently, it may not be a good fit for our training data.

### 3.1.2 Thompson Sampling

Thompson Sampling is a bandit algorithm which was first introduced in 1933. It involves assuming a simple prior distribution of the rewards of each arm.[11] Then, the algorithm plays the arm with the highest posterior probability of being the best arm. At each time step, it updates its knowledge of what is likely the best arm based on the reward received from the previous time step.[11] Despite its simplicity, Thompson Sampling has often achieved better empirical results than more complicated methods.[12]

### 3.1.3 Linear Thompson Sampling

Linear Thompson Sampling is an extension of Thompson Sampling designed for when a multi-armed bandit problem has linear payoff functions. It assumes a Gaussian distribution of the reward for each of the arms. The variance in this distribution, which would generally be calculated after all time steps have run, can instead be approximated at the current time step as follows:

$$v = r\sqrt{\frac{24}{\epsilon} d \log \frac{t}{\delta}},$$

where $r, \epsilon$, and $\delta$ are parameters passed into the function.[12] The algorithm keeps track of two matrices $\mathbf{A}$ and $\mathbf{b}$ which respectively contain data about the effects of various features on the reward, and a response vector based on the reward previously obtained. At each time step, the algorithm calculates the variance and samples from a multivariate normal distribution with mean $\mathbf{b}$ and covariance matrix $v^2\mathbf{A^{-1}}$. This calculation is repeated for each arm, and the best arm is chosen. Following that decision, the $\mathbf{A}$ and $\mathbf{b}$ matrices are updated based on the reward obtained.[12]

There are three different types of Linear Thompson Sampling used in this investigation. The "pooled" variant only keeps track of one $\mathbf{A}$ and $\mathbf{b}$ matrix for all arms. The "disjoint" variant stores a unique $\mathbf{A}$ and $\mathbf{b}$ matrix for each arm. Finally, the "hybrid" variant is an ensembled combination of the pooled and disjoint variants.

### 3.1.4 LinUCB

LinUCB, or Linear Upper Confidence Bound, is a contextual bandit algorithm that assumes that for every action, there is a linear relationship between the context and the reward. It models that relationship separately for each action.[9] LinUCB keeps track of two matrices $\mathbf{A}$ and $\mathbf{b}$ which, as for Linear Thompson Sampling, respectively contain data about the effects of features and a response vector. It computes the upper confidence bound of an arm and chooses the arm with the highest predicted reward. After choosing the arm, the reward obtained from that choice is used to update the existing $\mathbf{A}$ and $\mathbf{b}$ matrices.[9]

There are three different types of LinUCB algorithms used in this investigation. The disjoint variant is the same algorithm originally defined by Li *et al.* in 2012[9]. It stores separate $\mathbf{A}$ and $\mathbf{b}$ matrices for each arm. Conversely, pooled LinUCB assumes that all of the context for the student features, problem features, and all tutoring strategies can be used to map onto a reward function. As a result, it stores only one set of $\mathbf{A}$ and $\mathbf{b}$ matrices shared between all arms. We also use a version of LinUCB with hybrid linear models defined by Li *et al.* alongside disjoint LinUCB.[9] This version stores both shared $\mathbf{A}$ and $\mathbf{b}$ matrices (here known as the $\mathbf{A}_0$ and $\mathbf{b}_0$ matrices) as well as separate $\mathbf{A}$ and $\mathbf{b}$ matrices for each arm.[9]

## 3.2 Evaluation Procedure

To evaluate the performance of these models, we train them in a simulator using data from ASSISTments' TeacherASSIST program. This data contains instances of recommendations given to students by TeacherASSIST. Each data point contains the relevant problem features and student features at the time of recommendation, as well as whether the student answered the next problem correctly after receiving tutoring. More details of the exact features tracked in this data can be found in Prihar 2022.[13] Theoretically, if students tend to answer the next problem correctly after receiving a particular piece of tutoring, that tutoring is helpful. We want to train the algorithms to more often recommend tutoring strategies that helped students answer the next problem correctly. Previous literature has shown that any bandit algorithm can be accurately evaluated using previously recorded random traffic,[9] such as the kind recorded by TeacherASSIST.

### 3.2.1 Simulator

The simulator we created takes in a set of data from TeacherASSIST of past tutoring recommendations, along with the resulting "next problem correct" data. The simulation will repeat the following process a specified number of times, which we default to 100,000.

- Choose a random TeacherASSIST recommendation.
- Considering the recommendation as a bandit problem, request a bandit recommendation from the model being evaluated.
- If the model happens to output the same recommendation as TeacherASSIST, update the model according to whether the student answered the next problem correctly.

By repeating this process, we can train a model to obtain better recommendations over time. The detailed code of the simulation appears in appendix A.

### 3.2.2 Database

In order to store the results of the simulation, we created a local database in PostgreSQL that would store the histories of models' performance as well as their current state. To achieve this, we wrote a method for each model to save its state as a JSON string, as well as a method to load its state from a JSON string. Then, we created a database which, for each model, would store:

- A unique ID.
- The type of model (Random, Pooled LinUCB, Thompson Sampling, etc.).
- The model state as a JSON field.
- The history of the cumulative reward of the model at each time step.
- A boolean field representing if the model is active or not.

### 3.2.3 Grid Search

In order to evaluate the effect of the input parameters of each model, we performed a grid search using different values for those parameters. In each version of LinUCB, the input parameter $\alpha$ determines how the upper confidence bound will be scaled. A large scale means that exploration will be valued more than exploitation, as the confidence interval will have a wider range. Therefore, it is generally a measure of exploration vs. exploitation. For this parameter, we varied it by orders of magnitude from $2^{-3}, 2^{-2}, ..., 2^{2}, 2^{3}$.

In each version of Linear Thompson Sampling, the variables $r$, $\delta$, and $\epsilon$ affect the calculation of variance. The values of $\delta$ and $\epsilon$ are bounded between 0 and 1. In our investigation, we held $r$ at a value of 1 as it is a simple scaling factor. We also varied $\epsilon$ between values of 0.2 and 0.8. For the case of $\delta$, we held it at a constant value of 0.05, because the consistency of the regret bound of the algorithm depends on a probability of $1 - \delta$.[12] Higher values of $\delta$ don't significantly improve performance.

## 4 Results

After performing the evaluation outlined in section 3.2.3, we obtain the following data for each of our models.

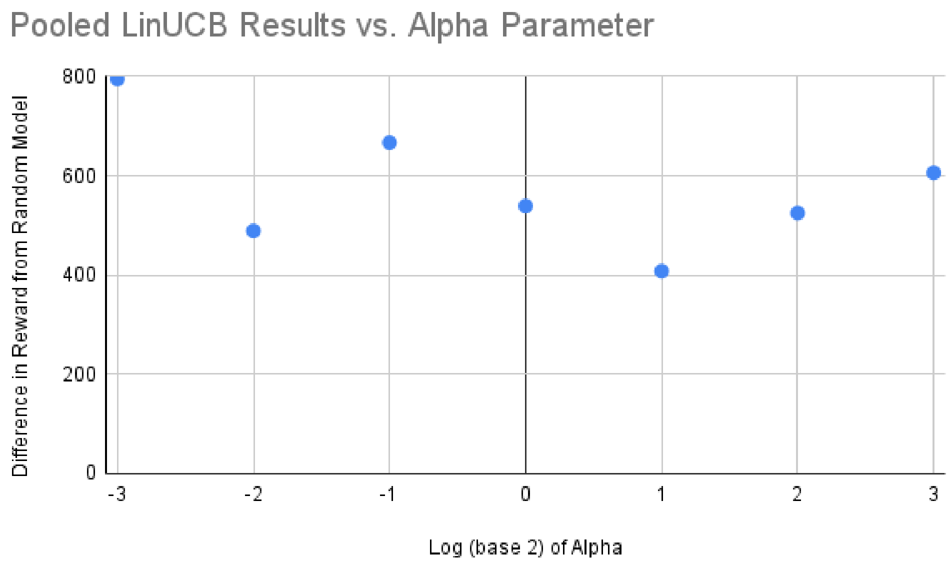


Figure 1: The results of running the simulation 100,000 times on Pooled LinUCB with different alpha values.

The Pooled LinUCB model obtained a maximum reward difference of 795 when compared to the random model. The graph in figure 1 indicates a slight negative correlation between the log of $\alpha$ and the difference in cumulative reward. Particularly, an $\alpha$ value of $\frac{1}{8}$ produced the highest reward.

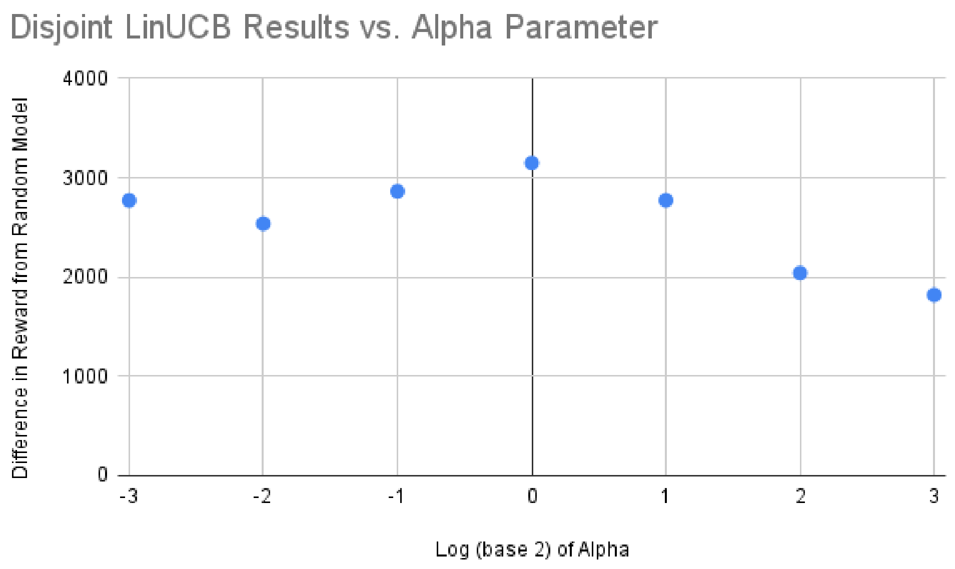


Figure 2: The results of running the simulation 100,000 times on Disjoint LinUCB with different alpha values.

The Disjoint LinUCB model obtained a maximum reward difference of 3149 when compared to the random model. The graph in figure 2 indicates a maximum around $\alpha = 1$, with models becoming worse as the value of $\alpha$ diverges from 1.

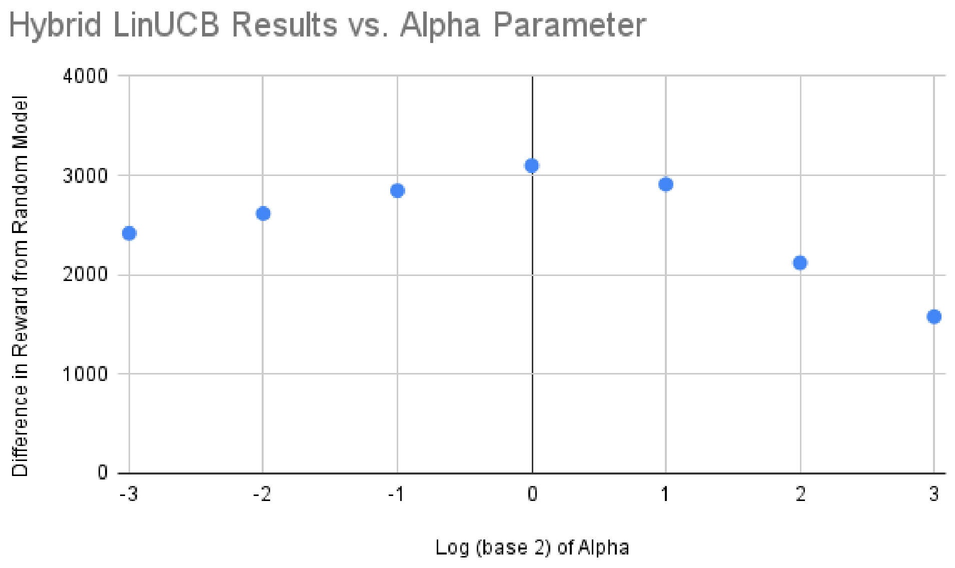


Figure 3: The results of running the simulation 100,000 times on Hybrid LinUCB with different alpha values.

The Hybrid LinUCB model obtained a maximum reward difference of 3099 when compared to the random model. The graph in figure 3 indicates a similar distribution to that of figure 2, but with more rapid changes as alpha diverges from 1. When $\alpha = 8$, the cumulative reward difference drops substantially to under 1700.

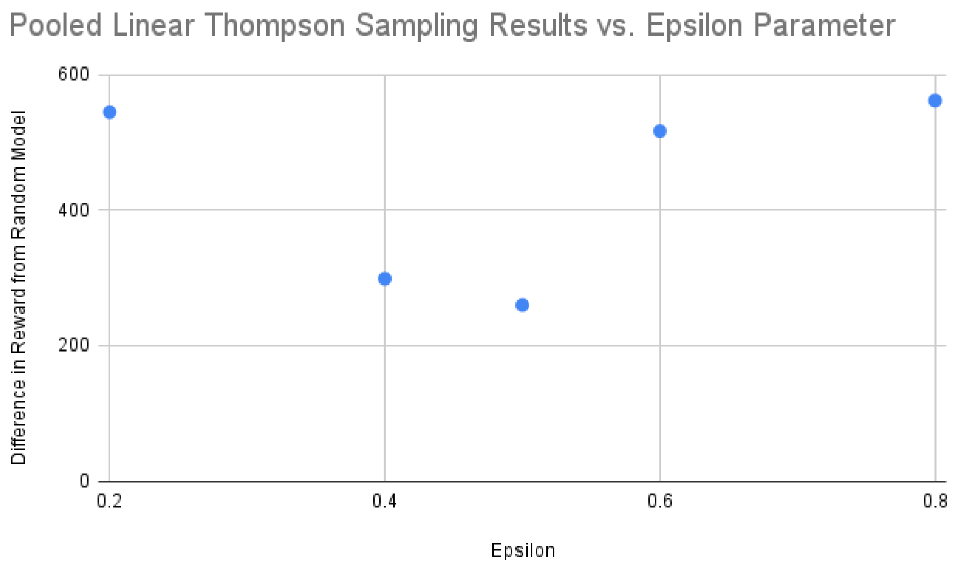


Figure 4: The results of running the simulation 100,000 times on Pooled Linear Thompson Sampling with different epsilon values.

The Pooled Linear Thompson Sampling model obtained a maximum reward of 562 when compared to the random model. The graph in figure 4 shows no clear relationship between epsilon and the cumulative reward difference.

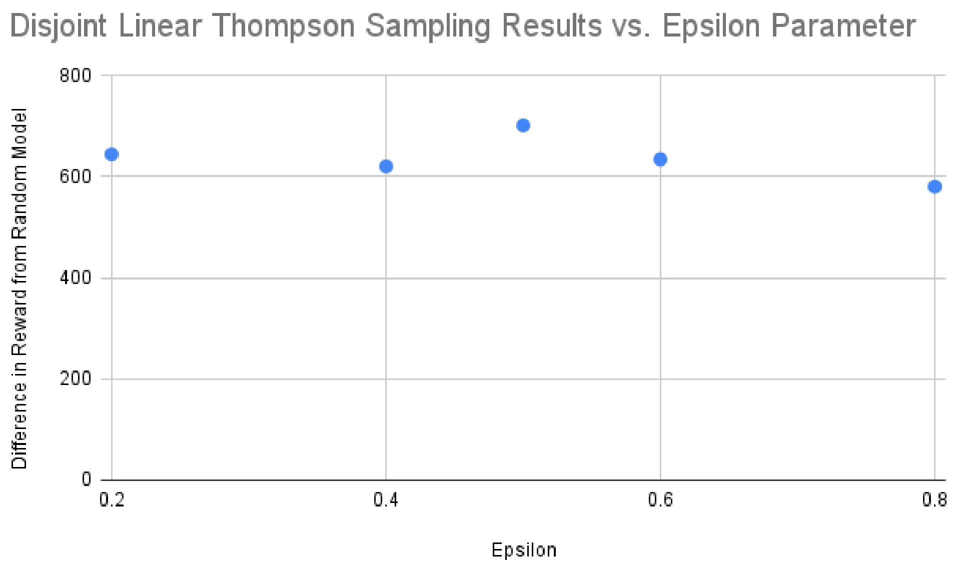


Figure 5: The results of running the simulation 100,000 times on Disjoint Linear Thompson Sampling with different epsilon values.

The Disjoint Linear Thompson Sampling model obtained a maximum reward of 701 when compared to the random model. The graph in figure 5 shows no clear relationship between epsilon and the cumulative reward difference. Compared to most other models, this model seems to be the most constant in cumulative reward difference.

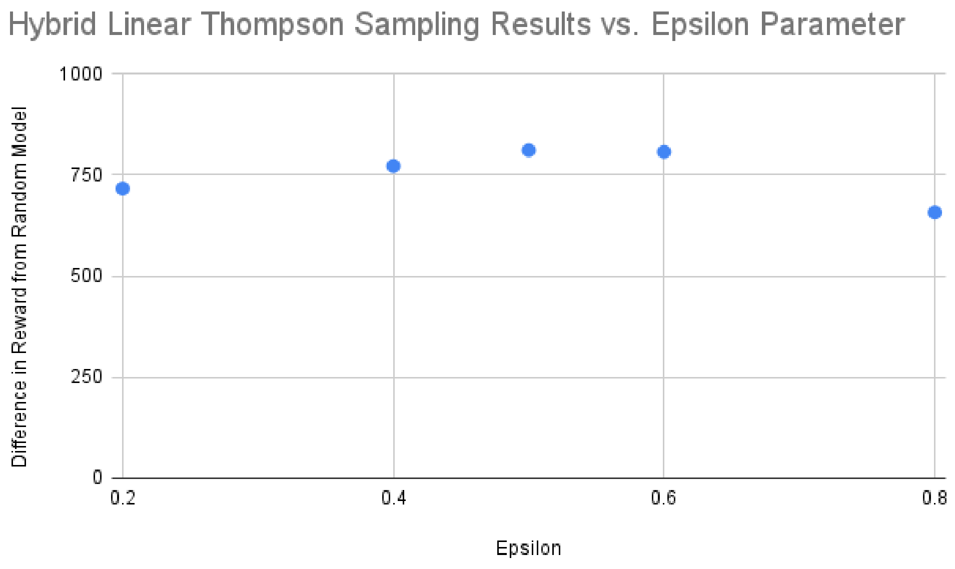
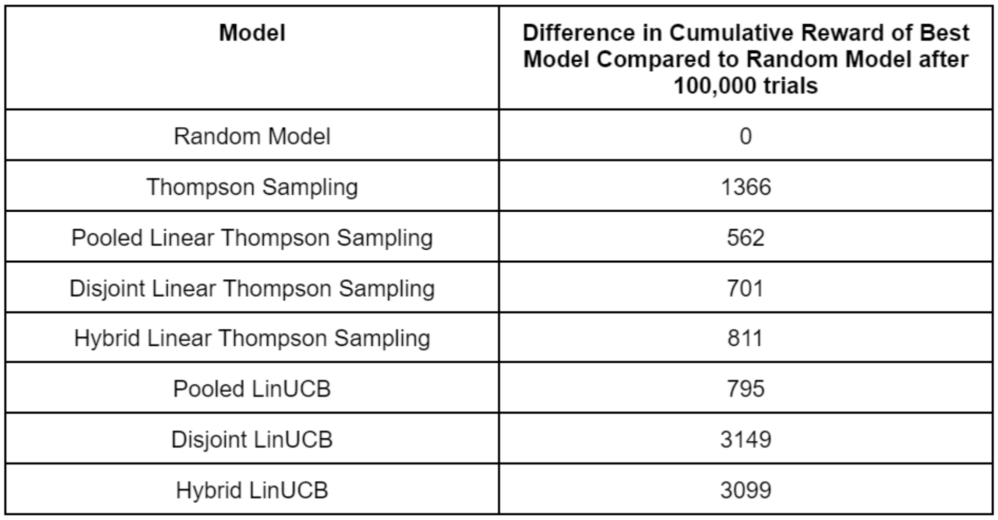


Figure 6: The results of running the simulation 100,000 times on Hybrid Linear Thompson Sampling with different epsilon values.

The Hybrid Linear Thompson Sampling model obtained a maximum reward of 811 when compared to the random model. The graph in figure 6 shows no clear relationship between epsilon and the cumulative reward difference.



| Model | Difference in Cumulative Reward of Best Model Compared to Random Model after 100,000 trials |
|---|---|
| Random Model | 0 |
| Thompson Sampling | 1366 |
| Pooled Linear Thompson Sampling | 562 |
| Disjoint Linear Thompson Sampling | 701 |
| Hybrid Linear Thompson Sampling | 811 |
| Pooled LinUCB | 795 |
| Disjoint LinUCB | 3149 |
| Hybrid LinUCB | 3099 |

Figure 7: Comparison of the best performing model for each model type when simulated 100,000 times.

Finally, in figure 7, we have a comparison of the best performing models for each model type. Disjoint and Hybrid LinUCB have significantly higher cumulative reward differences compared to other models. Thompson Sampling is the 3rd best algorithm listed, with cumulative reward difference of 1366. Interestingly, all forms of Linear Thompson Sampling performed worse than standard Thompson Sampling in these tests.

# 5 Discussion

While the Disjoint and Hybrid LinUCB models outperformed all the other models, the Hybrid LinUCB model did not outperform the Disjoint LinUCB model. This was unexpected, as it seemed like considering more shared data in Hybrid LinUCB would lead to better results. There are a few reasons why this may have occurred. Firstly, the shared features between tutoring strategies may not correlate to results as strongly as problem and student features. It may also be possible that the features we use for testing aren't as predictive as we anticipated. Hybrid LinUCB still performs well, but it may require some changes to outperform Disjoint LinUCB. As for Pooled LinUCB, it seems that the assumption that all context can be linearly mapped to a result is incorrect.

The Pooled and Disjoint Linear Thompson Sampling models performed better than the random model, but not as strongly as some of the other models. This could be due to our data not meeting the algorithms' requirements completely. Additionally, our Hybrid model, which ensembles the two other Linear Thompson Sampling models, performed slightly better than either individual model as expected.

Within the LinUCB models, it appears that changing the value of $\alpha$ away from 1 has a negative effect on the results, except in the case of Pooled LinUCB, where the best performance comes when $\alpha = \frac{1}{8}$. An $\alpha$ value of 1 indicates a balance between exploration and exploitation, which seems to produce the best results.

## 5.1 Limitations

These results are limited in that they come from a simulation, not real student data. It's important to simulate models like these before implementing them in student tutoring to avoid negative outcomes for the students, but this investigation could have benefited from some student tests. Additionally, our simulation may not be completely accurate to the real performance of students. Finally, we could have included more trials for each algorithm to determine more accurately how the parameters affect their performance.

## 5.2 Future Work

The results of this investigation will be used to improve the Reinforcement Learning Service of ASSISTments. Hybrid LinUCB will primarily be used to take advantage of the features shared across all tutor strategies. Disjoint LinUCB will also likely be implemented as a point of comparison. Additionally, a new set of shared features is being developed that may improve the performance of Hybrid LinUCB over Disjoint LinUCB.

## 6 Conclusion

This investigation has explored the use of various contextual bandit algorithms for recommending tutoring to students based on ASSISTments TeacherASSIST data. Disjoint LinUCB, Hybrid LinUCB, and Thompson Sampling were the 3 most effective bandit algorithms tested. An experiment testing each algorithm's parameters revealed some basic relationships between those parameters and the effectiveness of the algorithm. Particularly, Disjoint and Hybrid LinUCB perform best with an $\alpha$ value of 1, as exploration and exploitation are balanced. In the case of Pooled, Disjoint, and Hybrid Linear Thompson Sampling, the relationship between $\epsilon$ and algorithm performance is less clear. Further testing on the parameters of Linear Thompson Sampling would be helpful to clarify this relationship. Each of the models tested performed better on average than the random model, indicating that contextual bandit algorithms are indeed useful for recommending tutoring to students. Knowing this, ASSISTments can begin to implement using some of these models to actually perform recommendations to students. This may lead to improved student learning compared to current randomized tutoring recommendations.

## References

[1] A. Rafferty, H. Ying, and J. Williams, "Statistical consequences of using multi-armed bandits to conduct adaptive educational experiments," *Journal of Educational Data Mining*, vol. 11, p. 47–79, Jun. 2019.

[2] N. Heffernan and C. Heffernan, "The assistments ecosystem: Building a platform that brings scientists and teachers together for minimally invasive research on human learning and teaching.," *International Journal of Artificial Intelligence in Education*, vol. 24, pp. 470–497, 2014.

[3] T. Patikorn and N. T. Heffernan, "Effectiveness of crowd-sourcing on-demand assistance from teachers in online learning platforms," in *Proceedings of the Seventh ACM Conference on Learning @ Scale*, L@S '20, (New York, NY, USA), p. 115–124, Association for Computing Machinery, 2020.

[4] E. Prihar, T. Patikorn, A. Botelho, A. Sales, and N. Heffernan, "Toward personalizing students' education with crowdsourced tutoring," in *Proceedings of the Eighth ACM Conference on Learning @ Scale*, L@S '21, (New York, NY, USA), p. 37–45, Association for Computing Machinery, 2021.

[5] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," 2014.

[6] I. Osband and B. V. Roy, "Bootstrapped thompson sampling and deep exploration," 2015.

[7] C. Riquelme, G. Tucker, and J. Snoek, "Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling," 2018.

[8] J. Michael N. Katehakis, Arthur F. Veinott, "The multi-armed bandit problem: Decomposition and computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, 1987.

[9] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," *Proceedings of the 19th international conference on World wide web - WWW '10*, 2010.

[10] D. Agarwal, B.-C. Chen, and P. Elango, "Explore/exploit schemes for web content optimization, in '2009 ninth ieee international conference on data mining'," 2009.

[11] S. Agrawal and N. Goyal, "Analysis of thompson sampling for the multi-armed bandit problem," in *Proceedings of the 25th Annual Conference on Learning Theory* (S. Mannor, N. Srebro, and R. C. Williamson, eds.), vol. 23 of *Proceedings of Machine Learning Research*, (Edinburgh, Scotland), pp. 39.1–39.26, PMLR, 25–27 Jun 2012.

[12] S. Agrawal and N. Goyal, "Thompson sampling for contextual bandits with linear payoffs," in *Proceedings of the 30th International Conference on Machine Learning*, JMLR W&CP volume 28, (Atlanta, GA, USA), Journal of Machine Learning Research, 2013.

[13] E. Prihar, "Student support dataset description." https://osf.io/4nfvx/?show=view, 2022.

# Appendices

## A Simulator Code Excerpt

```
def simulation(args):

length, model_id, model, model_name, teacherassist_data, problem_features_data,
    user_features_data, tutor_strategy_features_data = args

records = []

t_1 = datetime.now()

simulation_count = 0
while simulation_count < length:
    # Pick a random teacherassist recommendation
    sample = teacherassist_data.sample(1).dropna(axis=1)
    assigned_tsid = sample['assigned_tutor_strategy_id'].iloc[0]

    # Postulate the recommendation as a bandit problem
    pid = sample['problem_id'].iloc[0]
    problem_features = problem_features_data.loc[pid]
    uid = sample['user_id'].iloc[0]
    user_features = user_features_data.loc[uid]
    option_features = {}
    for c in [c for c in sample.columns if 'tutor_strategy_id' in c]:
        tsid = int(sample[c].iloc[0])
        option_features[tsid] = tutor_strategy_features_data.loc[tsid]

    # Get a bandit recommendation
    bandit_recommendation = model.recommend(problem_features, user_features, option_features)

    # Update the bandit if the bandit recommendation was the same as TeacherASSIST's assignment
    if bandit_recommendation == assigned_tsid:
        reward = sample['next_problem_correctness'].iloc[0]
        model.update(problem_features, user_features, assigned_tsid,
            option_features[assigned_tsid], reward)
        simulation_count += 1

        # Record the recommendation
        records.append([model_id, reward])

return model, records
```

## B Model Code

### B.1 Random Model

```
class RandomModel:
def __init__(self, json_state):
    if json_state == "":
        self.model = {}
    else:
        self.model = json_state

def recommend(self, problem_features, student_features, option_features):
    return random.choice(list(option_features.keys()))

def update(self, problem_features, student_features, option_id, option_features, reward):
    if option_id not in self.model.keys():
        self.model[option_id] = {'alpha': 0, 'beta': 0}
    if reward == 1:
        self.model[option_id]['alpha'] += 1
    else:
        self.model[option_id]['beta'] += 1

def get_state(self, history):
    string_state = {}
    for int_key in self.model.keys():
        string_state[str(int_key)] = self.model[int_key]
    if history != "":
        string_state["cumulative_history"] = history
    return str(string_state).replace('\'', '"')
```

### B.2 Thompson Sampling Model

```
class ThompsonSamplingModel:
def __init__(self, json_state):
    if json_state == "":
        self.model = {}
    else:
        self.model = json_state

def recommend(self, problem_features, student_features, option_features):
    rewards = []
    for option in option_features.keys():
        if option in self.model.keys():
            alpha = self.model[option]['alpha']
            alpha += 1 if alpha == 0 else 0
            beta = self.model[option]['beta']
            beta += 1 if beta == 0 else 0
            rewards.append(rng.beta(alpha, beta))
        else:
            rewards.append(rng.beta(1, 1))
    return list(option_features.keys())[np.argmax(rewards)]
```

```
def update(self, problem_features, student_features, option_id, option_features, reward):
    if option_id not in self.model.keys():
        self.model[option_id] = {'alpha': 0, 'beta': 0}
    if reward == 1:
        self.model[option_id]['alpha'] += 1
    else:
        self.model[option_id]['beta'] += 1

def get_state(self, history):
    string_state = {}
    for option_id in self.model.keys():
        string_state[str(option_id)] = self.model[option_id]
    if history != "":
        string_state["cumulative_history"] = history
    return str(string_state).replace('\'', '"')
```

## B.3 Pooled Linear Thompson Sampling Model

```
class PooledLinearThompsonSamplingModel:
def __init__(self, json_state, r=1, delta=0.05, epsilon=0.5):
    if json_state == "":
        self.r = r
        self.delta = delta
        self.epsilon = epsilon
        self.big_a = None
        self.b = None
        self.t = 1
    else:
        self.r = json_state["r"]
        self.delta = json_state["delta"]
        self.epsilon = json_state["epsilon"]
        self.big_a = np.asarray(json_state["big_a"])
        self.b = np.asarray(json_state["b"])
        self.t = json_state["t"]

def recommend(self, problem_features, student_features, option_features):
    x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
    d = x.size

    v = self.r * np.sqrt((24 / self.epsilon) * d * np.log(self.t / self.delta))

    rewards = []
    for option_id in option_features.keys():
        if self.big_a is None:
            big_a = np.identity(d)
            b = np.zeros((d, 1))
        else:
            big_a = self.big_a
            b = self.b

        big_a_inv = np.linalg.inv(big_a)
        sample = np.random.multivariate_normal(b.flatten(), (v ** 2) * big_a_inv)
        reward = np.matmul(x.T, sample)
```

```
            rewards.append(reward)

        return list(option_features.keys())[np.argmax(rewards)]

    def update(self, problem_features, student_features, option_id, option_features, reward):
        x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
        d = x.size

        self.t = self.t + 1

        if self.big_a is None:
            big_a = np.identity(d)
            b = np.zeros((d, 1))
        else:
            big_a = self.big_a
            b = self.b

        big_a = big_a + np.matmul(x, x.T)
        b = b + reward * x

        self.big_a = big_a
        self.b = b

    def get_state(self, history):
        string_state = {"r": self.r, "delta": self.delta, "epsilon": self.epsilon, "t": self.t,
                        "big_a": self.big_a.tolist(), "b": self.b.tolist()}
        if history != "":
            string_state["cumulative_history"] = history
        return str(string_state).replace('\'', '"')
```

## B.4 Disjoint Linear Thompson Sampling Model

```
class DisjointLinearThompsonSamplingModel:
def __init__(self, json_state, r=1, delta=0.05, epsilon=0.5):
    if json_state == "":
        self.r = r
        self.delta = delta
        self.epsilon = epsilon
        self.big_a_dict = dict()
        self.b_dict = dict()
        self.t = 1

        # TODO temp count
        self.SVDErrorCount = 0
    else:
        self.r = json_state["r"]
        self.delta = json_state["delta"]
        self.epsilon = json_state["epsilon"]
        self.big_a_dict = {}
        self.b_dict = {}
        for key in json_state["big_a_dict"]:
            self.big_a_dict[key] = np.asarray(json_state["big_a_dict"][str(key)])
            self.b_dict[key] = np.asarray(json_state["b_dict"][str(key)])
        self.t = json_state["t"]

        # TODO temp count
        self.SVDErrorCount = 0

def recommend(self, problem_features, student_features, option_features):
    x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
    d = x.size

    # Can approximate v with current timestep instead of total timestep
    v = self.r * np.sqrt((24 / self.epsilon) * d * np.log(self.t / self.delta))

    # At every timestep we generate a sample from the distribution N(b, v^2 * big_a_inv),
    # then play the arm that maximizes that reward.

    rewards = []
    for option_id in option_features.keys():
        if option_id not in self.big_a_dict:
            big_a = np.identity(d)
            b = np.zeros((d, 1))
        else:
            big_a = self.big_a_dict[option_id]
            b = self.b_dict[option_id]

        big_a_inv = np.linalg.inv(big_a)
        try:
            sample = np.random.multivariate_normal(b.flatten(), (v ** 2) * big_a_inv)
            reward = np.matmul(x.T, sample)
            rewards.append(reward)
        except np.linalg.LinAlgError:
            self.SVDErrorCount += 1
            rewards.append(0)

    return list(option_features.keys())[np.argmax(rewards)]

def update(self, problem_features, student_features, option_id, option_features, reward):
    x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
    d = x.size

    self.t = self.t + 1

    if option_id not in self.big_a_dict:
        big_a = np.identity(d)
        b = np.zeros((d, 1))
    else:
        big_a = self.big_a_dict[option_id]
        b = self.b_dict[option_id]

    big_a = big_a + np.matmul(x, x.T)
    b = b + reward * x

    self.big_a_dict[option_id] = big_a
    self.b_dict[option_id] = b
```

```
def get_state(self, history):
    string_state = {"r": self.r, "delta": self.delta, "epsilon": self.epsilon, "t": self.t,
                    "big_a_dict": {}, "b_dict": {}}
    for option_id in self.big_a_dict.keys():
        string_state["big_a_dict"][str(option_id)] = self.big_a_dict[option_id].tolist()
        string_state["b_dict"][str(option_id)] = self.b_dict[option_id].tolist()
    if history != "":
        string_state["cumulative_history"] = history
    return str(string_state).replace('\'', '"')
```

## B.5 Hybrid Linear Thompson Sampling Model

```
class HybridThompsonSamplingModel:
def __init__(self, json_state, alpha=1, r=1, delta=0.05, epsilon=0.5):
    if json_state == "":
        self.alpha = alpha
        self.big_a_0 = np.array([])
        self.b_0 = np.array([])
        self.big_a_dict = dict()
        self.big_b_dict = dict()
        self.b_dict = dict()

        self.r = r
        self.delta = delta
        self.epsilon = epsilon
        self.t = 1
        self.SVDErrorCount = 0
    else:
        self.alpha = json_state["alpha"]
        self.big_a_0 = np.asarray(json_state["big_a_0"])
        self.b_0 = np.asarray(json_state["b_0"])
        self.big_a_dict = {}
        self.big_b_dict = {}
        self.b_dict = {}
        for key in json_state["big_a_dict"]:
            self.big_a_dict[key] = np.asarray(json_state["big_a_dict"][str(key)])
            self.big_b_dict[key] = np.asarray(json_state["big_b_dict"][str(key)])
            self.b_dict[key] = np.asarray(json_state["b_dict"][str(key)])

        self.r = json_state["r"]
        self.delta = json_state["delta"]
        self.epsilon = json_state["epsilon"]
        self.t = json_state["t"]
        self.SVDErrorCount = 0

def recommend(self, problem_features, student_features, option_features):
    x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
    d = x.size

    # Can approximate v with current timestep instead of total timestep
    v = self.r * np.sqrt((24 / self.epsilon) * d * np.log(self.t / self.delta))
    rewards = []
```

```
        k = list(option_features.values())[
            0].values.size  # all option_features elements have the same number of features
        if self.big_a_0.size == 0:
            big_a_0 = np.identity(k)
            b_0 = np.zeros((k, 1))
        else:
            big_a_0 = self.big_a_0
            b_0 = self.b_0

        big_a_0_inv = np.linalg.inv(big_a_0)

        for option_id in option_features.keys():
            z = option_features[option_id].values.reshape(-1, 1)

            if option_id not in self.big_a_dict:
                big_a = np.identity(d)
                b = np.zeros((d, 1))
            else:
                big_a = self.big_a_dict[option_id]
                b = self.b_dict[option_id]
            big_a_inv = np.linalg.inv(big_a)

            try:
                x_sample = np.random.multivariate_normal(b.flatten(), (v ** 2) * big_a_inv)
                x_reward = np.matmul(x.T, x_sample)

                z_sample = np.random.multivariate_normal(b_0.flatten(), (v ** 2) * big_a_0_inv)
                z_reward = np.matmul(z.T, z_sample)

                # z.T * beta_hat + x.T * theta_hat + alpha * sqrt(s)
                reward = z_reward + x_reward
                rewards.append(reward)
            except np.linalg.LinAlgError:
                self.SVDErrorCount += 1
                rewards.append(0)
        return list(option_features.keys())[np.argmax(rewards)]

    def update(self, problem_features, student_features, option_id, option_features, reward):
        x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
        d = x.size

        self.t = self.t + 1

        z = option_features.values.reshape(-1, 1)
        k = z.size

        if self.big_a_0.size == 0:
            big_a_0 = np.identity(k)
            b_0 = np.zeros((k, 1))
        else:
            big_a_0 = self.big_a_0
            b_0 = self.b_0

        if option_id not in self.big_a_dict:
            big_a = np.identity(d)
            big_b = np.zeros((d, k))
            b = np.zeros((d, 1))
        else:
            big_a = self.big_a_dict[option_id]
            big_b = self.big_b_dict[option_id]
            b = self.b_dict[option_id]

        big_a_inv = np.linalg.inv(big_a)

        big_a_0 = big_a_0 + np.matmul(big_b.T, np.matmul(big_a_inv, big_b))
        b_0 = b_0 + np.matmul(big_b.T, np.matmul(big_a_inv, b))
        big_a = big_a + np.matmul(x, x.T)
        big_b = big_b + np.matmul(x, z.T)
        b = b + reward * x
        big_a_0 = big_a_0 + np.matmul(z, z.T) - np.matmul(big_b.T, np.matmul(big_a_inv, big_b))
        b_0 = b_0 + reward * z - np.matmul(big_b.T, np.matmul(big_a_inv, b))

        self.big_a_0 = big_a_0
        self.b_0 = b_0
        self.big_a_dict[option_id] = big_a
        self.big_b_dict[option_id] = big_b
        self.b_dict[option_id] = b

    def get_state(self, history):
        string_state = {"alpha": self.alpha, "r": self.r, "delta": self.delta, "epsilon": self.epsilon,
                        "t": self.t, 'big_a_0': self.big_a_0.tolist(), 'b_0': self.b_0.tolist(),
                        "big_a_dict": {}, "big_b_dict": {}, "b_dict": {}}

        for option_id in self.big_a_dict.keys():
            string_state["big_a_dict"][str(option_id)] = self.big_a_dict[option_id].tolist()
            string_state['big_b_dict'][str(option_id)] = self.big_b_dict[option_id].tolist()
            string_state["b_dict"][str(option_id)] = self.b_dict[option_id].tolist()
        if history != "":
            string_state["cumulative_history"] = history
        return str(string_state).replace('\'', '"')
```

### B.6 Pooled LinUCB Model

```
    class PooledLinUCBModel:
    def __init__(self, json_state, alpha=1):
        if json_state == "":
            self.alpha = alpha
            self.big_a = None
            self.b = None
        else:
            self.alpha = json_state["alpha"]
            self.big_a = np.asarray(json_state["big_a"])
            self.b = np.asarray(json_state["b"])

    def recommend(self, problem_features, student_features, option_features):
        rewards = []
        for option in option_features.keys():
            x = pd.concat([problem_features, student_features, option_features[option]])
                .values.reshape(-1, 1)
            d = x.size
            if self.big_a is None:
                self.big_a = np.identity(d)
                self.b = np.zeros((d, 1))
            inv_big_a = np.linalg.inv(self.big_a)
            theta_hat = np.matmul(inv_big_a, self.b)
            rewards.append(np.matmul(theta_hat.T, x) + self.alpha * np.sqrt(np.matmul(np.matmul(x.T,
                inv_big_a), x)))
        return list(option_features.keys())[np.argmax(rewards)]

    def update(self, problem_features, student_features, option_id, option_features, reward):
        x = pd.concat([problem_features, student_features, option_features]).values.reshape(-1, 1)
        d = x.size
        if self.big_a is None:
            self.big_a = np.identity(d)
            self.b = np.zeros((d, 1))
        self.big_a = self.big_a + np.matmul(x, x.T)
        self.b = self.b + reward * x

    def get_state(self, history):
        string_state = {"alpha": self.alpha, "big_a": self.big_a.tolist(), "b": self.b.tolist()}
        if history != "":
            string_state["cumulative_history"] = history
        return str(string_state).replace('\'', '"')
```

## B.7 Disjoint LinUCB Model

```
    class DisjointLinUCBModel:
    def __init__(self, json_state, alpha=1):
        if json_state == "":
            self.alpha = alpha
            self.big_a_dict = {}
            self.b_dict = {}
        else:
            self.alpha = json_state["alpha"]
            self.big_a_dict = {}
            self.b_dict = {}
            for key in json_state["big_a_dict"]:
                self.big_a_dict[key] = np.asarray(json_state["big_a_dict"][str(key)])
                self.b_dict[key] = np.asarray(json_state["b_dict"][str(key)])

    def recommend(self, problem_features, student_features, option_features):
        x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
        d = x.size
        rewards = list()
        for option_id in option_features.keys():
            if option_id not in self.big_a_dict:
                big_a = np.identity(n=d)
                b = np.zeros(shape=(d, 1))
            else:
                # Get previously existing big_a and b
                big_a = self.big_a_dict[option_id]
                b = self.b_dict[option_id]

            big_a_inv = np.linalg.inv(big_a)
            theta_a = np.matmul(big_a_inv, b)

            expected_mean = np.matmul(theta_a.T, x)
            inner_product = np.matmul(np.matmul(x.T, big_a_inv), x)
            confidence_bound = self.alpha * np.sqrt(inner_product)
            reward = expected_mean + confidence_bound
            rewards.append(reward)
        return list(option_features.keys())[np.argmax(rewards)]

    def update(self, problem_features, student_features, option_id, option_features, reward):
        # Get features list
        x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
        d = x.size

        # Initialize new big_a and b if necessary
        if option_id not in self.big_a_dict:
            big_a = np.identity(n=d)
            b = np.zeros(shape=(d, 1))
            self.big_a_dict[option_id] = big_a
            self.b_dict[option_id] = b
        else:
            big_a = self.big_a_dict[option_id]
            b = self.b_dict[option_id]

        # Update big_a and b and store the results
        big_a = big_a + np.matmul(x, x.T)
        b = b + (reward * x)
        self.big_a_dict[option_id] = big_a
        self.b_dict[option_id] = b

    def get_state(self, history):
        string_state = {"alpha": self.alpha, "big_a_dict": {}, "b_dict": {}}

        for option_id in self.big_a_dict.keys():
            string_state["big_a_dict"][str(option_id)] = self.big_a_dict[option_id].tolist()
            string_state["b_dict"][str(option_id)] = self.b_dict[option_id].tolist()
        if history != "":
            string_state["cumulative_history"] = history
        return str(string_state).replace('\'', '"')
```

## B.8 Hybrid LinUCB Model

```
class HybridLinUCBModel:
def __init__(self, json_state, alpha=1):
    if json_state == "":
        self.alpha = alpha
        # self.big_a_0 = np.identity(k)   # shared big_a
        # self.b_0 = np.zeros((k, 1))     # shared b
        self.big_a_0 = np.array([])
```

```
            self.b_0 = np.array([])
            self.big_a_dict = dict()  # individual big_a’s
            self.big_b_dict = dict()  # individual big_b’s
            self.b_dict = dict()  # individual b’s
        else:
            self.alpha = json_state["alpha"]
            self.big_a_0 = np.asarray(json_state["big_a_0"])
            self.b_0 = np.asarray(json_state["b_0"])
            self.big_a_dict = {}
            self.big_b_dict = {}
            self.b_dict = {}
            for key in json_state["big_a_dict"]:
                self.big_a_dict[key] = np.asarray(json_state["big_a_dict"][str(key)])
                self.big_b_dict[key] = np.asarray(json_state["big_b_dict"][str(key)])
                self.b_dict[key] = np.asarray(json_state["b_dict"][str(key)])

    def recommend(self, problem_features, student_features, option_features):
        x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
        d = x.size
        rewards = []

        k = list(option_features.values())[
            0].values.size  # all option_features elements have the same number of features
        if self.big_a_0.size == 0:
            big_a_0 = np.identity(k)
            b_0 = np.zeros((k, 1))
        else:
            big_a_0 = self.big_a_0
            b_0 = self.b_0

        big_a_0_inv = np.linalg.inv(big_a_0)
        beta_hat = np.matmul(big_a_0_inv, b_0)

        for option_id in option_features.keys():
            z = option_features[option_id].values.reshape(-1, 1)

            if option_id not in self.big_a_dict:
                big_a = np.identity(d)
                big_b = np.zeros((d, k))
                b = np.zeros((d, 1))
            else:
                big_a = self.big_a_dict[option_id]
                big_b = self.big_b_dict[option_id]
                b = self.b_dict[option_id]
            big_a_inv = np.linalg.inv(big_a)
            theta_hat = np.matmul(big_a_inv, b - np.matmul(big_b, beta_hat))

            # z.T * big_a_0_inv * z
            s_term_1 = np.matmul(z.T, np.matmul(big_a_0_inv, z))
            # 2 * z.T * big_a_0_inv * big_b.T * big_a_inv * x
            s_term_2 = 2 * np.matmul(z.T, np.matmul(big_a_0_inv,
                np.matmul(big_b.T, np.matmul(big_a_inv, x))))
            # x.T * big_a_inv * x
            s_term_3 = np.matmul(x.T, np.matmul(big_a_inv, x))
            # x.T * big_a_inv * big_b * big_a_0_inv * big_b.T * big_a_inv * x
            s_term_4 = np.matmul(x.T,
                                 np.matmul(big_a_inv,
                                           np.matmul(big_b,
                                                     np.matmul(big_a_0_inv,
                                                               np.matmul(big_b.T,
                                                                         np.matmul(big_a_inv, x))))))

            s = s_term_1 - s_term_2 + s_term_3 + s_term_4

            # z.T * beta_hat + x.T * theta_hat + alpha * sqrt(s)
            reward = np.matmul(z.T, beta_hat) + np.matmul(x.T, theta_hat) + self.alpha * np.sqrt(s)
            rewards.append(reward)
        return list(option_features.keys())[np.argmax(rewards)]

    def update(self, problem_features, student_features, option_id, option_features, reward):
        x = pd.concat([problem_features, student_features]).values.reshape(-1, 1)
        d = x.size

        z = option_features.values.reshape(-1, 1)
        k = z.size

        if self.big_a_0.size == 0:
            big_a_0 = np.identity(k)
            b_0 = np.zeros((k, 1))
        else:
            big_a_0 = self.big_a_0
            b_0 = self.b_0

        if option_id not in self.big_a_dict:
            big_a = np.identity(d)
            big_b = np.zeros((d, k))
            b = np.zeros((d, 1))
        else:
            big_a = self.big_a_dict[option_id]
            big_b = self.big_b_dict[option_id]
            b = self.b_dict[option_id]

        big_a_inv = np.linalg.inv(big_a)

        big_a_0 = big_a_0 + np.matmul(big_b.T, np.matmul(big_a_inv, big_b))
        b_0 = b_0 + np.matmul(big_b.T, np.matmul(big_a_inv, b))
        big_a = big_a + np.matmul(x, x.T)
        big_b = big_b + np.matmul(x, z.T)
        b = b + reward * x
        big_a_0 = big_a_0 + np.matmul(z, z.T) - np.matmul(big_b.T, np.matmul(big_a_inv, big_b))
        b_0 = b_0 + reward * z - np.matmul(big_b.T, np.matmul(big_a_inv, b))

        self.big_a_0 = big_a_0
        self.b_0 = b_0
        self.big_a_dict[option_id] = big_a
        self.big_b_dict[option_id] = big_b
        self.b_dict[option_id] = b
```

```
def get_state(self, history):
    string_state = {"alpha": self.alpha, 'big_a_0': self.big_a_0.tolist(),
        'b_0': self.b_0.tolist(), "big_a_dict": {}, "big_b_dict": {}, "b_dict": {}}
    for option_id in self.big_a_dict.keys():
        string_state['big_a_dict'][str(option_id)] = self.big_a_dict[option_id].tolist()
        string_state['big_b_dict'][str(option_id)] = self.big_b_dict[option_id].tolist()
        string_state['b_dict'][str(option_id)] = self.b_dict[option_id].tolist()
    if history != "":
        string_state["cumulative_history"] = history
    return str(string_state).replace('\'', '"')
```