# CS 234 Winter 2022
# HW 5
# Due: Mar 11 at 11:59 pm (PST)

For submission instructions please refer to the website. For all problems, if you use an existing result from either the literature or a textbook to solve the exercise, you need to cite the source.

## 1 Offline Policy Evaluation [15 pts]

**Motivation** Previously, you have learned OPE techniques such as importance sampling and fitted Q evaluation (FQE). However, we can estimate value of a new policy in a much simpler form. The value of a new policy $\pi$ can be estimated as:

$$\rho(\pi) = \mathbb{E}_{(s,a)\sim d^{\mathcal{D}}, r\sim R(s,a)}\left[w_{\pi/\mathcal{D}}(s,a)\cdot r\right], \quad \omega_{\pi/\mathcal{D}}(s,a) = \frac{d^{\pi}(s,a)}{d^{\mathcal{D}}(s,a)}$$

This avoids the importance sampling calculation which has a high variance when the horizon $H$ is large, and function approximation error in FQE (if we use function approximation to learn the Q function). However, estimating the ratio $\omega_{\pi/\mathcal{D}}(s,a)$ is not easy because it contains state visit distributions: even if we can estimate $d^{\mathcal{D}}(s,a)$, we still have trouble estimating $d^{\pi}(s,a)$ without access to the MDP.

In the following problem, you'll view OPE as an optimization problem, where the solution to the optimization objective will be $\omega_{\pi/\mathcal{D}}(s,a)$.

**Problem** Consider an infinite-horizon, discounted MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma\rangle$ where $\gamma \in [0,1)$ and the state-action space is finite ($|\mathcal{S}\times\mathcal{A}| < \infty$). For any policy $\pi : S \to \Delta(A)$, recall that the discounted stationary state distribution is defined for any state $s \in S$ as

$$d^{\pi}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t \beta_t^{\pi}(s_t = s)$$

where $\beta_t^{\pi}(s_t = s)$ denotes the probability that the (random) state $s_t$ encountered by policy $\pi$ at timestep $t$ is equal to $s$:

$$\beta_t^{\pi}(s) := \Pr\left(s = s_t \mid s_0 \sim \beta, a_k \sim \pi(s_k), s_{k+1} \sim T(s_k, a_k) \text{ for } 0 \le k < t\right)$$

Let $\beta \in \Delta(S)$ be an initial state distribution such that $\beta_0^{\pi} = \beta(s)$ for all policies $\pi$ and any state $s \in S$. Recall that we came across discounted stationary state distributions in Assignment 2 while proving the Performance Difference lemma.

Let's consider the batch reinforcement-learning setting where we have a dataset D of transitions $(s, a, r, s')$ sampled in a completely unknown fashion. While $\mathcal{D}$ is not a policy, we will denote the distribution over state-action pairs sampled from $\mathcal{D}$ as $d^{\mathcal{D}}(s, a)$. For the remaining questions, define

$$d^\pi(s, a) = d^\pi(s)\pi(a \mid s)$$

(a). - **5 pts.** For any two arbitrary sets $\mathcal{X}$ and $\mathcal{Y}$, we denote the class of all bounded functions mapping from $\mathcal{X}$ to $\mathcal{Y}$ as $\{\mathcal{X} \to \mathcal{Y}\}$. Let $x \in \{S \times A \to \mathbb{R}\}$ such that for all $(s, a) \in S \times A$,

$$J(x) = \frac{1}{2}\mathbb{E}_{(s,a)\sim d^{\mathcal{D}}}\left[x(s, a)^2\right] - \mathbb{E}_{(s,a)\sim d^\pi}[x(s, a)].$$

Let $x^\star$ be the minimizer of $J(x)$. Prove that $x^\star(s, a) = \omega_{\pi/\mathcal{D}}$ for any state-action pair $(s, a) \in S \times A$.

**Hint**: $J(x)$ is of the form $\frac{1}{2}nx^2 - mx$ and we want to find the $x$ that minimizes $J(x)$.

Expanding out, we have

$$\begin{aligned} J(x) &= \frac{1}{2}\mathbb{E}_{(s,a)\sim d^{\mathcal{D}}}\left[x(s, a)^2\right] - \mathbb{E}_{(s,a)\sim d^\pi}[x(s, a)] \\ &= \sum_{s,a} \frac{d^D(s, a)}{2} x(s, a)^2 - d^\pi(s, a)x(s, a). \end{aligned} \tag{1}$$

Now, if we can individually – that is, on an $(a, s)$ by $(a, s)$ basis – minimize

$$\frac{d^D(s, a)}{2} x(s, a)^2 - d^\pi(s, a)x(s, a),$$

we will minimize the entire sum. Now, by a standard quadratic + derivative argument, we have that for constants $n, m$

$$\arg\min_z \frac{1}{2}nz^2 - mz = \frac{m}{n},$$

which we get from

$$nz - m = 0.$$

As such, it is evident that for an individual $(a, s)$, we will have

$$\arg\min_{x(s,a)} \frac{d^D(s, a)}{2} x(s, a)^2 - d^\pi(s, a)x(s, a) = \frac{d^\pi(s, a)}{d^D(s, a)}.$$

At this point, we've minimized one (arbitrary) term in the sum in (1); if we do it for *every* $(a, s)$ tuple, we will minimize the entirety of (1)

$$\sum_{s,a} \frac{d^D(s, a)}{2} x(s, a)^2 - d^\pi(s, a)x(s, a),$$

and thus minimize $J$. Thus,

$$x(s, a) = \frac{d^\pi(s, a)}{d^D(s, a)}$$

minimizes (1), via minimization on a term-by-term basis of the sum.

(b). - **5 pts** If we can solve the optimization problem from part (a), then we can find $\omega_{\pi/D}$. But the objective function in the previous part is problematic since the second term requires samples from the stationary state-action distribution $d^\pi(s,a)$. Consider a value function $\nu \in \{S \times A \to \mathbb{R}\}$ whose rewards are given by some $x \in \{S \times A \to \mathbb{R}\}$ such that for all $(s,a) \in S \times A$,

$$\nu(s,a) \coloneqq x(s,a) + \gamma \mathbb{E}_{s' \sim \mathcal{T}(\cdot|s,a)}\left[\mathbb{E}_{a' \sim \pi(\cdot|s')}\left[\nu\left(s',a'\right)\right]\right]$$

Prove that $\mathbb{E}_{(s,a)\sim d^\pi}[x(s,a)] = (1-\gamma)\mathbb{E}_{s_0 \sim \beta}\left[\mathbb{E}_{a_0 \sim \pi(\cdot|s_0)}\left[\nu\left(s_0,a_0\right)\right]\right]$.

**Hint**: Use $\beta_t$, the state visitation probability at step $t$ when following $\pi$.

Begin with the LHS. Expanding out, we have

$$\begin{aligned}
\mathbb{E}_{(s,a)\sim d^\pi}[x(s,a)] &= \sum_s \sum_a d^\pi(s,a)x(s,a) \\
&= \sum_s \sum_a d^\pi(s)\pi(a|s)x(s,a) \\
&= \sum_s \sum_a (1-\gamma)\sum_{t=0}^{\infty} \gamma^t B_t^\pi(s_t = s)\pi(a|s)x(s,a) \\
&= (1-\gamma)\sum_{t=0}^{\infty}\sum_s \sum_a \gamma^t B_t^\pi(s_t = s)\pi(a|s)x(s,a) \\
&= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t \sum_{s'_t}\sum_{a'_t} B_t^\pi(s_t = s'_t)\pi(a'_t|s'_t)x(s'_t,a'_t) \\
&= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t E_{s'_t,a'_t\sim d_t^\pi}[x(s'_t,a'_t)] \\
&= (1-\gamma)\left[E_{s'_0,a'_0\sim d_0^\pi}[x(s'_0,a'_0)] + \sum_{t=1}^{\infty}\gamma^t E_{s'_t,a'_t\sim d_t^\pi}[x(s'_t,a'_t)]\right] \\
&= (1-\gamma)E_{s'_0,a'_0\sim d_0^\pi}\left[x(s'_0,a'_0) + \sum_{t=1}^{\infty}\gamma^t E_{s'_t,a'_t\sim d_t^\pi}[x(s'_t,a'_t)|s'_0,a'_0]\right] \\
&= (1-\gamma)E_{s'_0,a'_0\sim d_0^\pi}\left[x(s'_0,a'_0) + \gamma E_{s'_1\sim\mathcal{T}(\cdot|s'_0,a'_0)}E_{a'_1\sim\pi(\cdot|s'_1)}v(s'_1,a'_1)\right] \\
&= (1-\gamma)E_{s'_0,a'_0\sim d_0^\pi}[v(s'_0,a'_0)] \\
&= (1-\gamma)E_{s'_0\sim\beta}E_{a'_0\sim\pi(\cdot|s'_0)}[v(s'_0,a'_0)].
\end{aligned}$$

The result follows from expansion of an expectation; substitution of the definition of $d^\pi$; linearity of expectations; towering; the recursive definition of $v$; and more linearity of expectation.

(c). - **5 pts** The result from part (b) allows us to remove the expectation under $d^\pi(s,a)$ in $J(x)$. Rewrite $J(x)$ as $J(\nu)$ using the result from part (b). Your objective should no longer have the expectation under $d^\pi(s,a)$. You should simplify your answer by using the zero-reward Bellman operator:

$$\mathcal{B}^\pi \nu(s,a) = \gamma \mathbb{E}_{s'\sim T(s,a), a'\sim\pi(s')}\left[\nu\left(s',a'\right)\right].$$

Combined with your result from part (a), what is the minimizer of $J(\nu)$?

Using the result above, we have

$$\begin{aligned}
J(v) &= \frac{1}{2}\mathbb{E}_{(s,a)\sim d^{\mathcal{D}}}\left[v(s,a)^2\right] - \mathbb{E}_{(s,a)\sim d^{\pi}}[v(s,a)] \\
&= \frac{1}{2}\mathbb{E}_{(s,a)\sim d^{\mathcal{D}}}\left[v(s,a)^2\right] - (1-\gamma)\mathbb{E}_{s_0\sim\beta}\left[\mathbb{E}_{a_0\sim\pi(\cdot|s_0)}\left[\nu\left(s_0,a_0\right)\right]\right] \\
&= \frac{1}{2}\mathbb{E}_{(s,a)\sim d^{\mathcal{D}}}\left[v(s,a)^2\right] - (1-\gamma)\mathbb{E}_{(s,a)\sim\langle\beta,\pi\rangle}\left[\nu(s,a)\right]
\end{aligned}$$

Then, we apply our proof in a.) (moving the $(1-\gamma)$ multiplier "onto" the probability in the minimization in a.))

$$v^*(s,a) = \frac{\pi(a|s)\beta(s)(1-\gamma)}{\delta^D(s,a).}$$

## 2 RL for personalized recommendations [60pts]

One of the most influential applications of RL in the real-world is in generating personalized recommendations for videos, movies, games, music, etc. Companies such as Netflix (?), Spotify (?), Yahoo (?) and Microsoft (?) use contextual bandits to recommend content that is more likely to catch the user's attention. Generating recommendations is an important task for these companies — the value of Netflix recommendations to the company itself is estimated at \$1 billion (?).
Content recommendations take place in a dynamical system containing feedback loops (?) affecting both users and producers. Reading recommended articles and watching recommended videos changes people's interests and preferences, and content providers change what they make as they learn what their audience values. However, when the system recommends content to the user, the user's choices are strongly influenced by the set of options offered. This creates a feedback loop in which training data reflects *both* user preferences and previous algorithmic recommendations.
In this problem, we will investigate how *video creators* learn from people's interactions with a recommendation system for videos, how they change the videos they produce accordingly, and what these provider-side feedback loops mean for the users. Dynamics similar to the ones we investigate here have been studied in newsrooms as journalists respond to real-time information about article popularity (?), and on YouTube as video creators use metrics such as clicks, minutes watched, likes and dislikes, comments, and more to determine what video topics and formats their audience prefers (?).
We have created a (toy) simulation that allows you to model a video recommender system as a contextual bandit problem.[1] In our simulation, assume we have a certain fixed number of users $N_u$. Each user has a set of preferences, and their preference sets are all different from one another. We start off with some number of videos we can recommend to these users. These videos correspond to the arms of the contextual bandit. Initially there are $N_a$ arms. Your goal is to develop a contextual bandit algorithm that can recommend the best videos to each user as quickly as possible.
In our Warfarin setting from assignment 4, $N_a$ was fixed: we always chose from three different dosages for all patients. However, video hosting and recommendation sites like YouTube are more

[1] According to ?, YouTube does not currently use RL for their recommendations, but other video recommendation systems do, as noted above.

dynamic. Content creators monitor user behavior and add new videos (i.e. arms) to the platform in response to the popularity of their past videos. In other words, $N_a$ keeps increasing over time.
How does this change the problem to be solved? Are we still in the bandit setting or is this now morphing into an RL problem? For now, we will treat it as a bandit problem. Remember that the number of users is static: $N_u$ is a constant and doesn't change. In the coding portion of this assignment, you will study the effect of adding new arms into the contextual bandit setup, the different strategies we can employ to add these arms and measure how they affect performance.

## Implementational details of the simulator

Most of the simulator has been written for you but the details might be useful in analysing your results. The only parts of the simulator you will need to write are the different strategies used to add more arms to the contextual bandit.
The simulator is initialized with $N_u$ users and $N_a$ arms where each user and arm is represented as a feature vector of dimension $d$. Each element of these vectors is initialized i.i.d from a normal distribution. When reset, the simulator returns a random user context vector from the set of $N_u$ users. When the algorithm chooses an arm, the simulator returns a reward of 0 if the arm chosen was the best arm for that user and -1 otherwise.
We will be running the simulator for $T$ steps where each step represents one user interaction. After every $K$ steps, we add an arm to the simulator using one of three different strategies outlined below.
Go through the code for the simulator in `sim.py`. Most of the simulator is implemented for you: the only method you will need to implement is `update_arms()`.

## Implementing the Bandit algorithm [10pts]

For this assignment we will be using the Disjoint Linear Upper Confidence Bound (LinUCB) algorithm from **?**. You have already implemented this in assignment 4 but you will have to make some minor changes to your code to account for the fact that the number of arms could keep increasing now. You will need to write about 15 lines of code in `hw5.py` for this part, most of which should come directly from assignment 4. The hints provided in `hw5.py` should help you with this. This has been implemented. It is leveraged in the following sections

## Implementing the arm update strategies [30pts]

We will now implement three different strategies to add arms to our simulator. Each arm is associated with its true feature vector $\theta^*_a$. This is the $d$-dimensional feature vector we assigned to each arm when we initialized the simulator. This is the $\theta$ the LinUCB algorithm is trying to learn for each arm through $A$ and $b$. When we create new arms, we need to create these new feature vectors as well.
When coming up with strategies to add arms, we need to put ourselves in the shoes of content creators and think about how we want to optimize for the videos that go up on our channels. When making such decisions, we only consider the previous $K$ steps since we added the last arm. Consider the three strategies outlined below:

1. **Popular:** For the last $K$ steps, pick the two most popular arms and create a new arm with the mean of the true feature vectors of these two arms. For example, assume $a_1$ and $a_2$ were the two most chosen arms in the previous $K$ steps with true feature vectors $\theta^*_1$ and $\theta^*_2$ respectively. Now create a new arm $a$ with $\theta^*_a = \frac{\theta^*_1+\theta^*_2}{2}$.

In the real world, this is similar to a naive approach where content creators create a new video based on their two most recommended videos from the last month.

2. **Corrective:** For the last $K$ steps, consider all the users for whom we made incorrect recommendations. Assume we know what the best arm would have been for each of those users. Consider taking corrective action by creating a new arm that is the weighted mean of all these true best arms for these users. For example, say for the last $K$ steps, we got $n_1 + n_2$ predictions wrong where the true best arm was $a_1$ $n_1$ times and $a_2$ $n_2$ times. Create a new arm $a$ with $\theta_a^* = \frac{n_1\theta_1^* + n_2\theta_2^*}{n_1+n_2}$.
   In the real world, this is analogous to content creators adapting their content to give their viewers what they want to watch based on feedback from viewers about their preferences.

3. **Counterfactual:** Consider the following counterfactual: For the previous $K$ steps, had there existed an additional arm $a$, what would its true feature vector $\theta_a^*$ have to be so that it would have been better than the chosen arm at each of those $K$ steps? There are several ways to pose this optimization problem. Consider the following formulation:

$$\theta_a^* = \arg\max_\theta \frac{1}{2}\sum_{i=1}^{K}(\theta^T x_i - \theta_i^T x_i)^2$$

   Here $x_i$ is the context vector of the user at step $i$. $\theta_i$ is the true feature vector of the arm chosen at step $i$. We can now optimize this objective using batch gradient ascent.

$$\theta_a \leftarrow \theta_a + \eta\frac{\partial L}{\partial \theta}$$

   Here $\eta$ is the learning rate and $L = \sum_{i=1}^{K}(\theta^T x_i - \theta_i^T x_i)^2$.
   We can find $\frac{\partial L}{\partial \theta}$ directly as $\sum_{i=1}^{K}(\theta^T x_i - \theta_i^T x_i)x_i$. We can write the update rule as

$$\theta_a \leftarrow \theta_a + \eta\sum_{i=1}^{K}(\theta_a^T x_i - \theta_i^T x_i)x_i$$

   In the real world, this is akin to asking the question, "What item could I have recommended in the past $K$ steps that would have been better than all recommendations made in the past $K$ steps?" In asking this, the creator aims to produce a new video that would appeal to all users more than the video that was recommended to them.

Implement these three methods in the `update_arms()` function in `sim.py`. This should be about 25 lines of code. Hints have been provided in the form of comments. **Each method is worth 10 points.**

Code implemented!

## Analysis [15pts]

For all the experiments in this section, we will initialize the simulator with 25 users and 10 arms each represented by a feature vector of dimension 10. We will run the simulation for a total of 10000

steps and use $\alpha = 1.0$ for the LinUCB algorithm. All these arguments have already been set for you and you will not have to change them.
For the questions below, please answer with just **2-3 sentences**.

1. **[1pts]** Run the LinUCB algorithm without adding any new arms. Run the algorithm for 5 different seeds. Report the mean and standard deviation of the total fraction correct. You can do this by running the following command:

   ```
   python hw5.py -s 0 1 2 3 4 -u none
   ```

   This method returned an average fraction correct of .565, with standard deviation .074. Results are presented in Figure 1 below (see Appendix).

2. **[2pts]** Run the LinUCB algorithm by adding new arms with the **popular** strategy. Run it with 4 different values of $K \in 500, 1000, 2500, 5000$. Run each $K$ for 5 different seeds. Report the mean and standard deviation of the total fraction correct for each $K$. You can do this by running the following command:

   ```
   python hw5.py -s 0 1 2 3 4 -u popular -k 500 1000 2500 5000
   ```

   Are your results better or worse than the results when you didn't add any new arms? Why do you think this is the case? Results are presented in Figure 2 below. As we see, this method performs better as $K$ increases – ultimately, at $K = 5000$ achieving a fraction correct of .552 and .078. In other words, even for pretty big $K$, the popular method held up worse than the 'None' method. Reasons for this could be one or some combination of:

   (a) Popular – particularly in the non-stationary internet setting, where things are always evolving – is unlikely to be suboptimal. As such, we're introducing a bunch of (likely) suboptimal bandits, and then pulling them a bunch (due to UCB's exploration); hence we get suboptimal performance.

   (b) Averaging the features of the two most popular choices may result in a "vanilla" and non-informative feature. This may not be worth adding an arm for.

   (c) In general, it may not be worth adding an arm (only muddies the waters/obfuscates things) unless the arm is *that* good/worthwhile. And here, the new features, as derived from popularity, may not be all that worthwhile – hence, we could just be throwing useless arms into the bandit, resulting in inferior performance.

3. **[2pts]** Run the LinUCB algorithm by adding new arms with the **corrective** strategy. Run it with 4 different values of $K \in 500, 1000, 2500, 5000$. Run each $K$ for 5 different seeds. Report the mean and standard deviation of the total fraction correct for each $K$. You can do this by running the following command:

   ```
   python hw5.py -s 0 1 2 3 4 -u corrective -k 500 1000 2500 5000
   ```

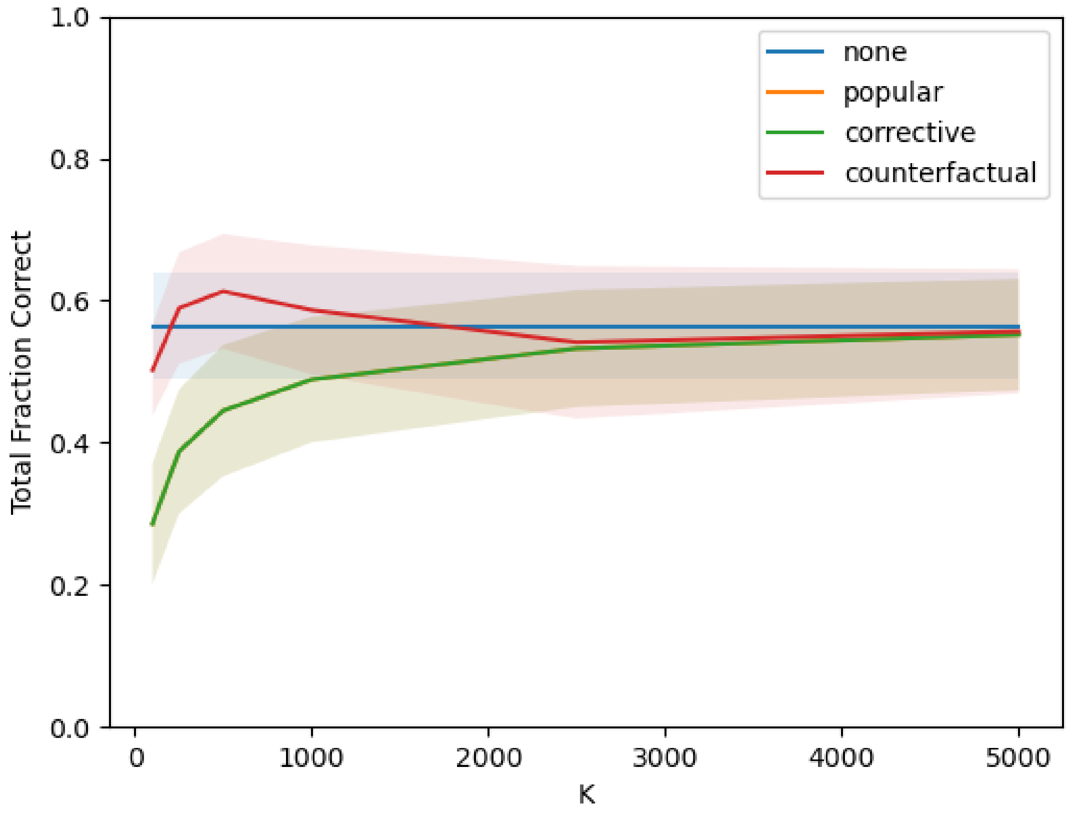
Which arm update strategy is better – corrective or popular? Or are they the same? Why do you think this is the case? As shown in Figure 3, we see that results are nearly identical. I suspect that this is because in the early stages, there is considerable overlap between the popular choices and the incorrect choices – hence, each new arm takes similar features. In turn, and a similar sequence of arms gets pulled for the rest of the bandit sequence, resulting in the observed results.

4. **[5pts]** Run the LinUCB algorithm by adding new arms with the **counterfactual** strategy. Run it with 4 different values of $K \in 500, 1000, 2500, 5000$. Run each $K$ for 5 different seeds. Report the mean and standard deviation of the total fraction correct for each $K$. You can do this by running the following command:

```
python hw5.py -s 0 1 2 3 4 -u counterfactual -k 500 1000 2500 5000
```

   Plot a table to compare the results from all 3 arm update strategies and when you don't add new arms for different values of $K$. Which arm update strategy is the best of the three? Which of them perform better than the case where we don't add new arms? Why do you think this is the case?Results are shown in Figure 4. We see that the counterfactual method outperforms the the other arm updates at all attempted $K$ values, as averaged over the same five seeds. Importantly, the counterfactual method holds a larger lead for smaller $K$ values (e.g. .613 counterfactual vs. .445 popular/corrective at $K = 500$, but .556 counterfactual at $K = 5000$ vs .552 popular/corrective at $K = 5000$), but this lead appears to narrow as K increases. This behavior is visualized over more $K$ in Figure 6 as well; however, the four values of $K$ compared above are visualized below.

   As for why, I'd venture to guess that the counterfactual holds up better than the popular/corrective method for reasons outlined above: whereas the popular/corrective method is likely adding sub-optimal arms, the counterfactual (even with it's one gradient step) is adding an arm that has at least taken one step in the direction of improving things. Hence, the improved performance.



5. **[2pts]** Now run all the algorithms together for $K = 500$ and plot a graph of the fraction incorrect over time. You can run this with the following command:

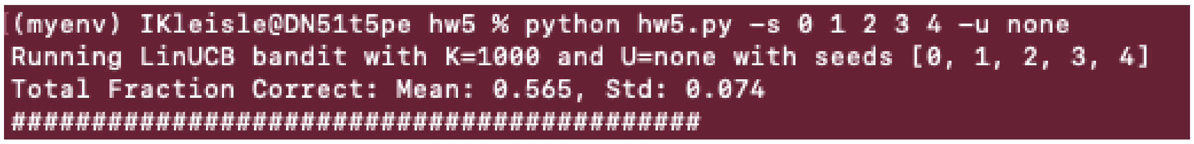
```
python hw5.py -s 0 1 2 3 4 -u none popular corrective counterfactual -k 500
--plot-u
```

How do the different strategies perform for $K = 500$? As shown in Figure 5, we see that initially, with fewer users seen, the don't-add-arms method (None) seems to have the fewest incorrect, with corrective close by. However, as more users are seen, the counterfactual begins to pull ahead, achieving a smaller fraction incorrect after just over 3,000 users seen. This lead is maintained thereafter.

6. **[3pts]** Let us analyse the effect of $K$ on all our algorithms. Plot a graph of the total fraction correct for all the algorithms for $K \in \{100, 250, 500, 1000, 2000, 3000, 4000, 5000\}$. You can run this with the following command:

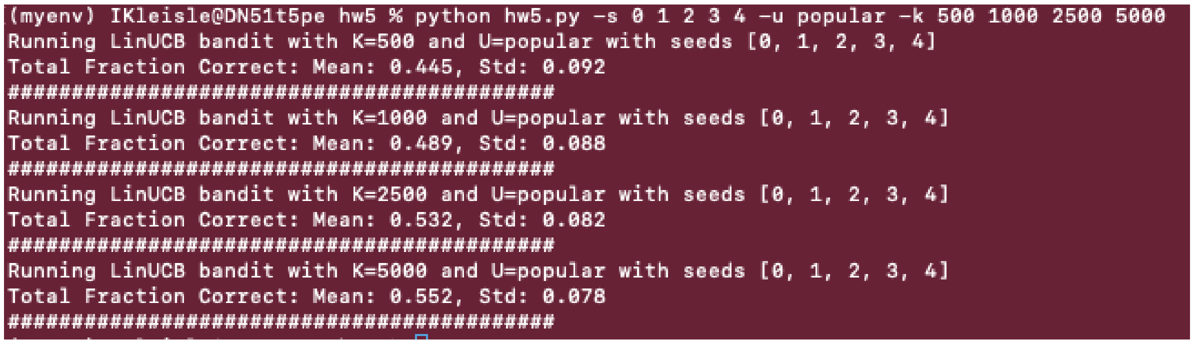
```
python hw5.py -s 0 1 2 3 4 -u none popular corrective counterfactual -k 100 250
500 1000 2000 3000 4000 5000 --plot-k
```

Which strategies get better as you increase $K$ and which strategies get worse? Why do you think this is the case?Looking at Figure 6, we see that the counterfactual holds the lead over the other two methods up until about $K = 2000$, when None takes the lead (corrective and popular appear worst throughout) thereafter. In the end (i.e. around $K = 4000, 5000$, corrective, popular, and counterfactual recover to just below None (with counterfactual "best-of-the-rest"), and at $K = 5000$, they're about comparable. Importantly, counterfactual looks like it gets worse between 2000 and 3000, at which point it begins its recovery. As for why, I'd again guess that the counterfactual holds up better than the popular/corrective method for reasons outlined above: whereas the popular/corrective method is likely adding sub-optimal arms, the counterfactual (even with it's one gradient step) is adding an arm that has at least taken one step in the direction of improving things. Hence, the improved performance. However, the no-addition method also appears to hold up okay here – perhaps via "the simple model is often the best one" type argument. And all generally converge on the same fraction incorrect for large $K$, demonstrating a WLLN-esque behavior – that is, with a sufficiently large sample, they'll all get close to the same thing.



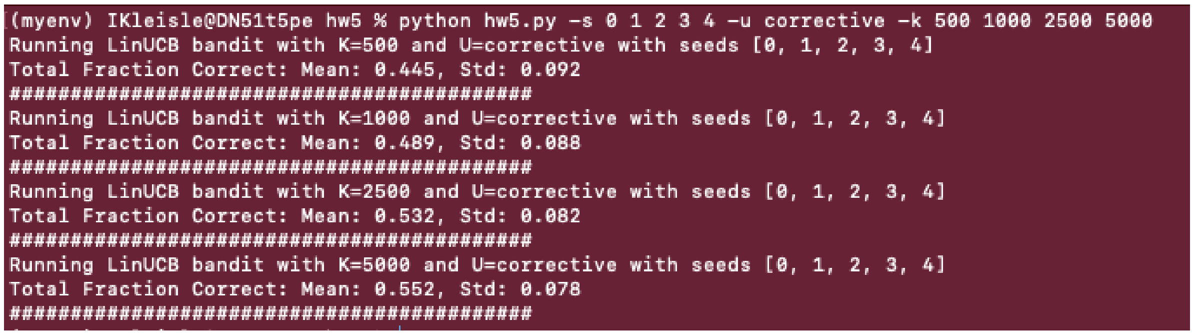
```
(myenv) IKleisle@DN51t5pe hw5 % python hw5.py -s 0 1 2 3 4 -u none
Running LinUCB bandit with K=1000 and U=none with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.565, Std: 0.074
###########################################
```

Figure 1: Run 2A (No Additions)



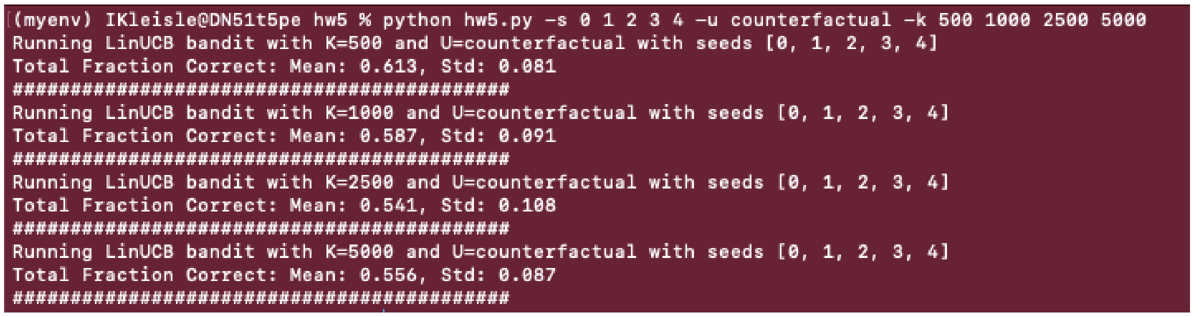
```
(myenv) IKleisle@DN51t5pe hw5 % python hw5.py -s 0 1 2 3 4 -u popular -k 500 1000 2500 5000
Running LinUCB bandit with K=500 and U=popular with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.445, Std: 0.092
###########################################
Running LinUCB bandit with K=1000 and U=popular with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.489, Std: 0.088
###########################################
Running LinUCB bandit with K=2500 and U=popular with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.532, Std: 0.082
###########################################
Running LinUCB bandit with K=5000 and U=popular with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.552, Std: 0.078
###########################################
```

Figure 2: Run 2B (Popular)



```
(myenv) IKleisle@DN51t5pe hw5 % python hw5.py -s 0 1 2 3 4 -u corrective -k 500 1000 2500 5000
Running LinUCB bandit with K=500 and U=corrective with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.445, Std: 0.092
###########################################
Running LinUCB bandit with K=1000 and U=corrective with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.489, Std: 0.088
###########################################
Running LinUCB bandit with K=2500 and U=corrective with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.532, Std: 0.082
###########################################
Running LinUCB bandit with K=5000 and U=corrective with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.552, Std: 0.078
###########################################
```

Figure 3: Run 2C (Corrective)



```
(myenv) IKleisle@DN51t5pe hw5 % python hw5.py -s 0 1 2 3 4 -u counterfactual -k 500 1000 2500 5000
Running LinUCB bandit with K=500 and U=counterfactual with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.613, Std: 0.081
###########################################
Running LinUCB bandit with K=1000 and U=counterfactual with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.587, Std: 0.091
###########################################
Running LinUCB bandit with K=2500 and U=counterfactual with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.541, Std: 0.108
###########################################
Running LinUCB bandit with K=5000 and U=counterfactual with seeds [0, 1, 2, 3, 4]
Total Fraction Correct: Mean: 0.556, Std: 0.087
###########################################
```

Figure 4: Run 2D (Counterfactual)

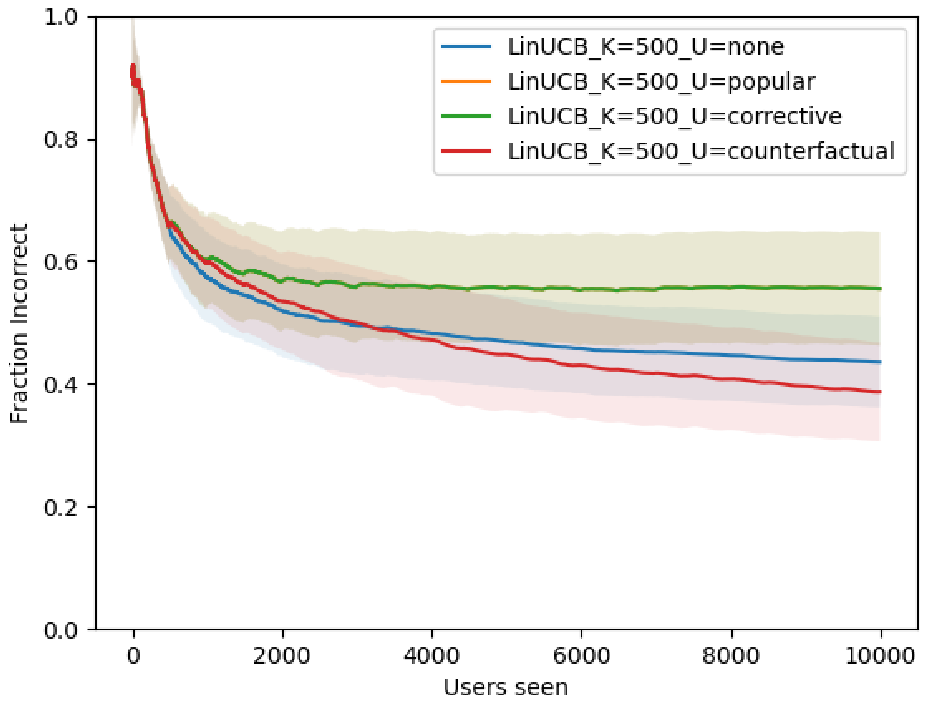


Figure 5: Run 2E (Comparison)

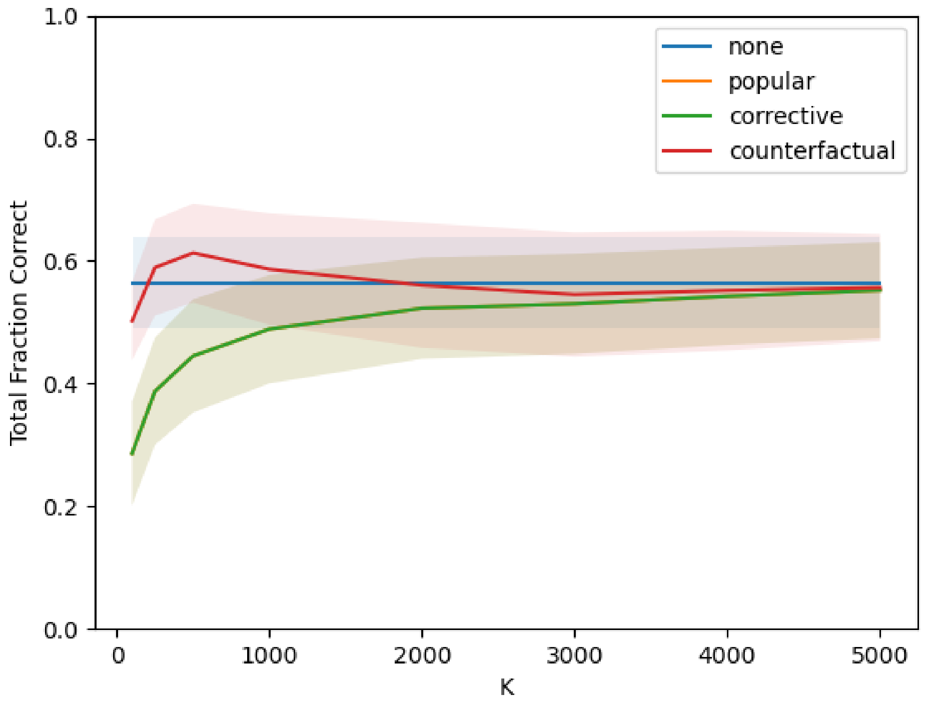


Figure 6: Run 2F (K-Analysis)

## Discussion [5pts]

7. We now would like you to think about the different interactions between content creators, users, and AI algorithms.

   Please look at the accompanying material in the Assignment release Ed post and consider the following (non-exhaustive) possible scenarios.

   (a) **User preferences are static**: if at time $t_1$ a user only likes to watch videos about dogs, at time $t_{100}$ he/she/they will still only want to watch videos about dogs. As creators learn more about user preferences, they cooperate in making more of the content that the users prefer and are rewarded in the metrics for doing so. AI algorithms also provide recommendations of content most likely to align with the static user preferences.

   (b) In a second situation, user **meta-preferences (e.g. for popular content) are static but the AI algorithm can contribute to feedback loops**. For example, assume some users always prefer popular content, but their specific content preferences will shift over time depending on what appears to be most popular in the content they are exposed to. Initially, automated algorithm randomly selects some content to show more frequently. Seeing that it has been viewed more frequently, content creators create more of that content. The automated algorithm then serves it to more people, causing its popularity to increase. Creators specializing in initially popular content are rewarded, causing a "rich get richer" phenomenon, both at the level of individual content items, and for particular creators.

   (c) Consider an example: users who did not previously like cereal are influenced by an advertisement or by repeated exposure to low-view-count cereal videos. They come to prefer cereal and will now click on cereal videos in the future, but may be unaware that the algorithm influenced this shift in preferences. In this third scenario, **users' preferences are dynamic** and are influenced and changed by the content served. This can both influence what later content is recommended by the AI algorithm and which future new content is created by content creators.

   (d) A more complicated scenario can be when users may change in response to perceived or actual algorithmic classification. For example, a user may click on many items relating to Taylor Swift. This may result in an AI algorithm serving them many future items related to Taylor swift or implicitly categorizing them as "a Taylor Swift fan." Although the user had not previously realized their own preference for Taylor Swift, once the user sees how much Taylor Swift content they are now being served, the user realizes that the AI algorithm models them as someone who really likes Taylor Swift. The user may react to this perceived labeling of being a "Taylor Swift fan", potentially changing their own behavior either to embrace this classification (and click more on Taylor Swift content) or rejecting this categorization (and click less on future Taylor Swift content). Here users' preferences are dynamic and **users react to their algorithmic** funneling into different soft categories, creating "looping kinds".

   **[2pts]** i. Consider the four situations (of many others) above. Which of these do you think may have problematic ethical implications? What information would be needed to understand which situation is happening in a particular platform-users-content creator universe? Do you think it is likely that platform creators or content creators have access to this information in their standard logs, or do you expect specific experiments would be needed? (2-4 sentences) In my view, c.) and d.) have the capacity to be most ethically problematic. The case of c.) is pretty straightforward: replace "cereal" with something more extremist, and you have the NY Times article discussed in class, where users may be pushed towards more extremist and potentially violent views. As has been repeatedly evidenced over the past zero to ten years (both domestically and internationally), pushing individuals towards extremism online gets people hurt. In terms of access to logs, and/or knowledge and complicity, I think the record is pretty straightforward there as well: nefarious users obviously knew of their misdeeds, and the platforms themselves have often been spineless in addressing these loops. And platform creators (who collect large sums of metadata) had access to logs of this behavior (if not, that is a failure of the platform) – such has been the subject of much reporting, congressional testimony, and litigation. Whether in the context of right-wing extremism, foreign influence in elections, terrorist recruiting, or elsewhere, it is well-understood that many platforms were aware of their problems, but insufficiently responsive to them.

In my view, d.) may also raise ethical issues. While the Taylor swift example is largely innocuous, I find the idea of "labeling" users online slightly unsettling. In particular, many youth/teenagers – who may still be developing emotionally, or be generally vulnerable – are very "online" today. And I think labeling that sub-population in this way may have unhealthy consequences for this population, which makes me uneasy.

**[3pts]** ii. Who is responsible for identifying a negative feedback loop and intervening to break it: content creators or platforms? Justify your answer with a 3-6 sentence explanation that references one or more of the above scenarios. In my view, a number of parties have responsibilities when it comes to identifying a negative feedback loop and intervening. For the purposes of the discussion here, let's run with example c.); however, let's assume it's not "cereal" videos, and instead something more nefarious, such as the radicalization videos discussed in class. Responsible parties include:

- Principally, the platform creators should sufficiently monitor their platforms, and swiftly intervene when dangerous content swarms their platform. In my judgement, these creators wear the bulk of the responsibility: when inventing the platforms, they know there will be bad actors. They should be prepared to counter them. Moreover, even when harmful negative feedback loops inadvertently arise (e.g. the platform made a good faith effort to prevent), platforms should be swift in taking action. In many ways, I can understand an algorithm starting to go off the rails when made in good faith – however, when brought to the platform's intervention, the platform should quickly intervene.
- Users/content creators should behave in morally/ethically responsible ways. Of course, this will not always be the case – we all have the obligation to be a good person, but of course some individuals will still elect not to.
- The press, whistleblowers, watchdogs, and other third parties should be vocal when they identify a negative feedback loop. For example, the New York Times should run features on Youtube's right-wing radicalization feedback loop without hesitation; similarly, ProPublica should continue to investigate Facebook's role in the American milita movement, etc. And whistleblowers who expose these feedback loops should not face retaliation.

In short, negative feedback loops may be unavoidable, but platforms, content producers, and the broader public should flag them without hesitation, and the platforms should take swift action to close the loops.