XCS234 Assignment 5

Due Sunday, October 26 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs234-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. For SCPD classes, it is also important that students avoid opening pull requests containing their solution code on the shared assignment repositories. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing these files, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do NOT make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and can be run locally.

- `hidden`: These unit tests are NOT visible locally. These hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned. These tests will evaluate elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

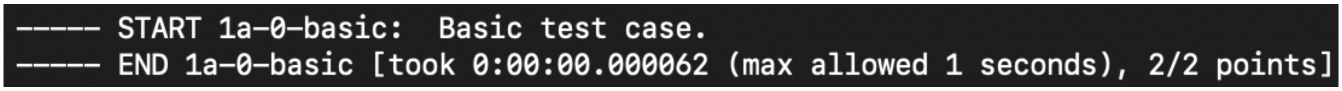
Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

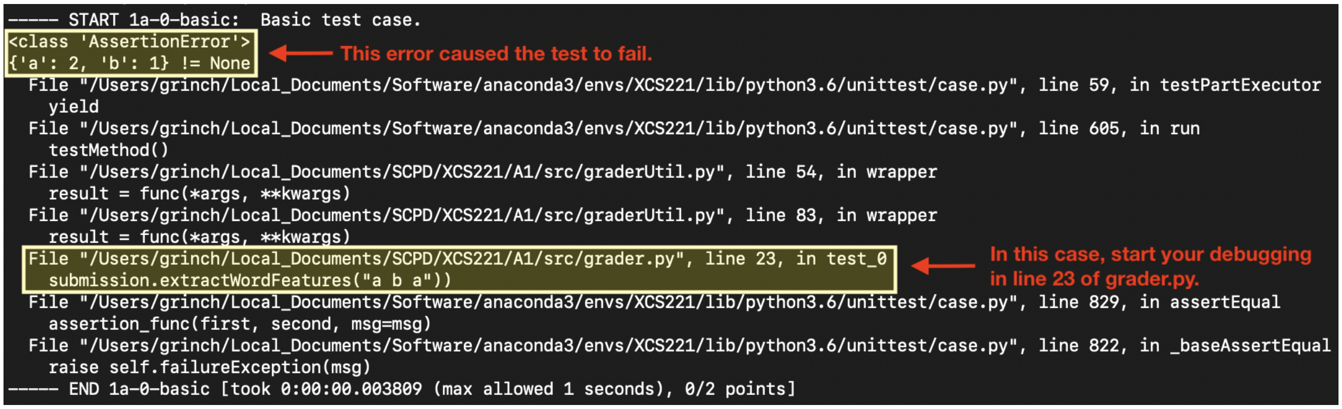
Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:



```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:



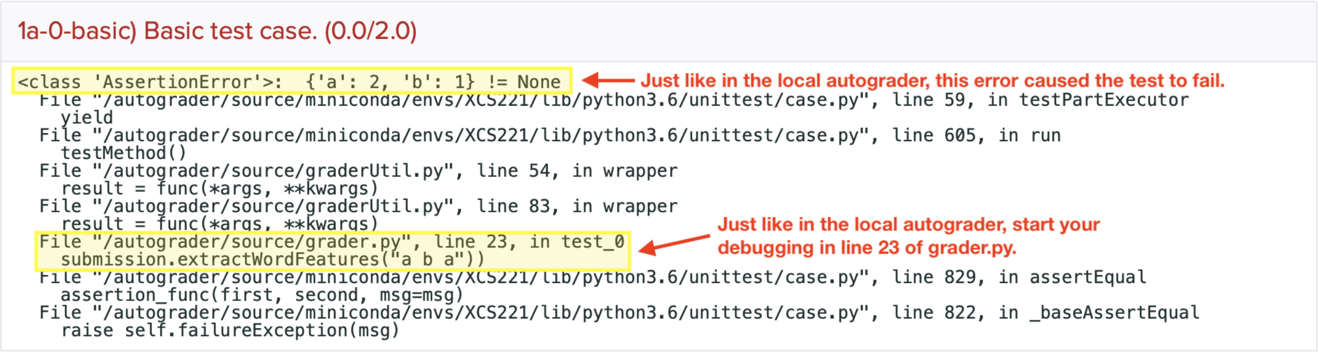
```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

This error caused the test to fail.

In this case, start your debugging in line 23 of grader.py.

Remote Execution

Basic and hidden tests are treated the same by the remote autograder, however the output of hidden tests will only appear once you upload your code to GradeScope. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

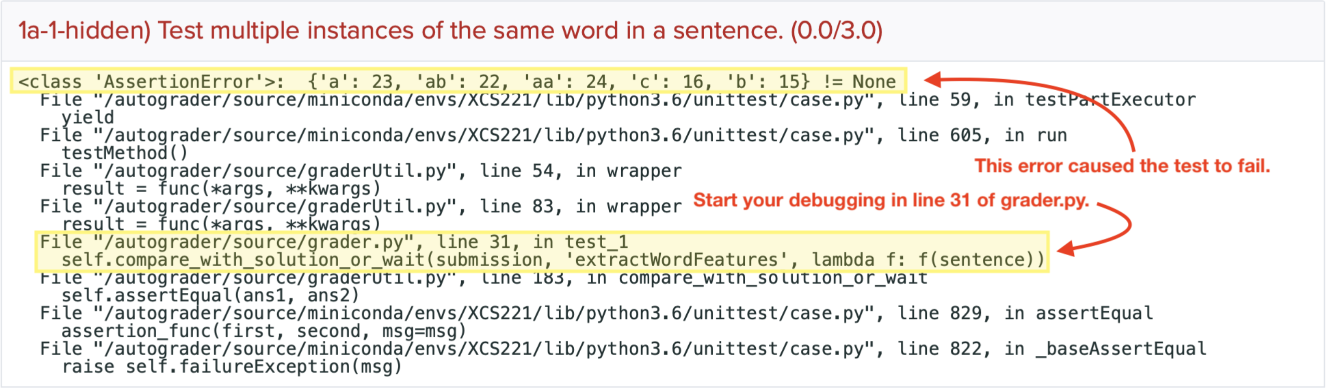


1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.



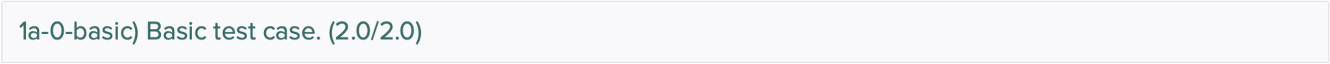
1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

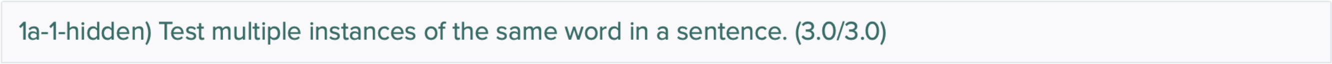
This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.



1a-0-basic) Basic test case. (2.0/2.0)



1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

0 Introduction

In this assignment, you will get the opportunity to apply what we have learnt in class about multi-armed bandit problems to a real-world application. In this case, we will be predicting the appropriate dosage for the drug Warfarin that we should prescribe to a patient given a set of features describing the patient.

As part of this real-world application we will mimic implementations provided in the following papers:

1. Linear Disjoint Upper Confidence Bound: A Contextual-Bandit Approach to Personalized News Article Recommendation
2. Thompson Sampling for Contextual Bandits: Thompson Sampling for Contextual Bandits with Linear Payoffs

Following this practical application we will continue to investigate multi-armed bandit algorithms and their applications in the real world through personalized recommendatation systems.

Files to submit:

1. `src/submission/submission.py`

1 Estimation of the Warfarin Dose

Warfarin

Warfarin is the most widely used oral blood anticoagulant agent worldwide; with more than 30 million prescriptions for this drug in the United States in 2004. The appropriate dose of warfarin is difficult to establish because it can vary substantially among patients, and the consequences of taking an incorrect dose can be severe. If a patient receives a dosage that is too high, they may experience excessive anti-coagulation (which can lead to dangerous bleeding), and if a patient receives a dosage which is too low, they may experience inadequate anti-coagulation (which can mean that it is not helping to prevent blood clots). Because incorrect doses contribute to a high rate of adverse effects, there is interest in developing improved strategies for determining the appropriate dose (see paper for further details).

Commonly used approaches to prescribe the initial warfarin dosage are the *pharmacogenetic algorithm* developed by the IWPC (International Warfarin Pharmacogenetics Consortium), the *clinical algorithm* and a *fixed-dose* approach.

In practice a patient is typically prescribed an initial dose, the doctor then monitors how the patient responds to the dosage, and then adjusts the patient's dosage. This interaction can proceed for several rounds before the best dosage is identified. However, it is best if the correct dosage can be initially prescribed.

This question is motivated by the challenge of Warfarin dosing, and considers a simplification of this important problem, using real data. The goal of this question is to explore the performance of multi-armed bandit algorithms to best predict the correct dosage of Warfarin for a patient *without* a trial-an-error procedure as typically employed.

Problem setting

Let T be the number of time steps. At each time step t, a new patient arrives and we observe its individual feature vector $X_t \in \mathbb{R}^d$: this represents the available knowledge about the patient (e.g., gender, age, ...). The decision-maker (your algorithm) has access to K arms, where the arm represents the warfarin dosage to provide to the patient. For simplicity, we discretize the actions into $K = 3$

- Low warfarin dose: under 21mg/week
- Medium warfarin dose: 21-49 mg/week
- High warfarin dose: above 49mg/week

If the algorithm identifies the correct dosage for the patient, the reward is 0, otherwise a reward of -1 is received.

Lattimore and Szepesvári have a nice series of blog posts that provide a good introduction to bandit algorithms, available here: BanditAlgs.com. The Introduction and the Linear Bandit posts may be particularly of interest. For more details of the available Bandit literature you can check out the Bandit Algorithms Book by the same authors.

Dataset

We use a publicly available patient dataset that was collected by staff at the Pharmacogenetics and Pharmacogenomics Knowledge Base (PharmGKB) for 5700 patients who were treated with warfarin from 21 research groups spanning 9 countries and 4 continents. You can find the data in `warfarin.csv` and metadata containing a description of each column in `metadata.xls`. Features of each patient in this dataset includes, demographics (gender, race, ...), background (height, weight, medical history, ...), phenotypes and genotypes.

Importantly, this data contains the true patient-specific optimal warfarin doses (which are initially unknown but are eventually found through the physician-guided dose adjustment process over the course of a few weeks) for 5528 patients. You may find this data in mg/week in `Therapeutic Dose of Warfarin`[1] column in `warfarin.csv`. There are in total 5528 patient with known therapeutic dose of warfarin in the dataset (you may drop and ignore the remaining 173 patients for the purpose of this question). Given this data you can classify the right dosage for each patient as *low*: less than 21 mg/week, *medium*: 21-49 mg/week and *high*: more than 49 mg/week, as defined in paper and our Problem Setting section.

Note: the data processing steps are already implemented for you in `utils/data_preprocessing.py`

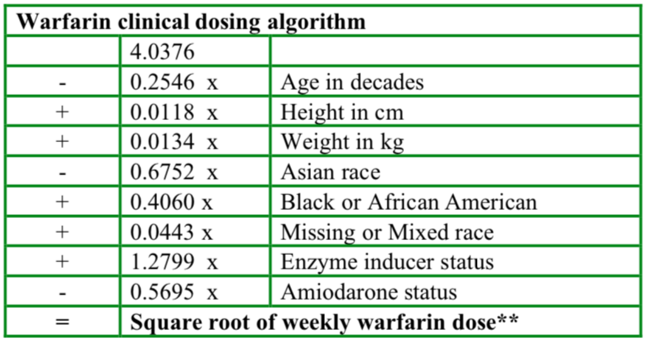
[1] You cannot use `Therapeutic Dose of Warfarin` data as an input to your algorithm.

(a) **[4 points (Coding)]**

Please implement the following two baselines in `submission.py`.

(a) *Fixed-dose*: This approach will assign 35mg/week (medium) dose to all patients.

(b) *Warfarin Clinical Dosing Algorithm*: This method is a linear model based on age, height, weight, race and medications that patient is taking. You can find details of the exact model below which is taken from section S1f of `data/appx.pdf`.



Warfarin clinical dosing algorithm		
	4.0376	
-	0.2546 x	Age in decades
+	0.0118 x	Height in cm
+	0.0134 x	Weight in kg
-	0.6752 x	Asian race
+	0.4060 x	Black or African American
+	0.0443 x	Missing or Mixed race
+	1.2799 x	Enzyme inducer status
-	0.5695 x	Amiodarone status
=	**Square root of weekly warfarin dose****	

Figure 1: Definition of clinical (linear) model take from section 1f of appx.pdf

Run the fixed dosing algorithm and clinical dosing algorithm with the following commands respectively:

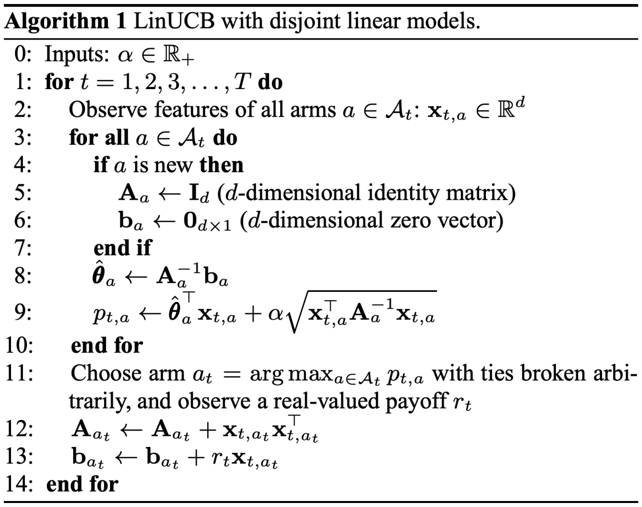
```
$ python run.py --model fixed
$ python run.py --model clinical
```

You should see the total_fraction_correct to be fixed at about 0.61 for fixed dose and 0.64 for clinical dose algorithm.

Hint: Look into section 1f of appx.pdf for a description of the features used in the clincal model

(b) **[16 points (Coding)]**

Please implement the Disjoint Linear Upper Confidence Bound (LinUCB) algorithm from paper in `submission.py`. Below we have provided a snippet of Algorithm 1 from the listed paper. For a thorough description of terms please read through the paper.



Algorithm 1 LinUCB with disjoint linear models.

```
Inputs: $\alpha \in \mathbb{R}_+$
for $t = 1, 2, 3, \ldots, T$ do
  Observe features of all arms $a \in \mathcal{A}_t$: $\mathbf{x}_{t,a} \in \mathbb{R}^d$
  for all $a \in \mathcal{A}_t$ do
    if $a$ is new then
      $\mathbf{A}_a \leftarrow \mathbf{I}_d$ ($d$-dimensional identity matrix)
      $\mathbf{b}_a \leftarrow \mathbf{0}_{d\times 1}$ ($d$-dimensional zero vector)
    end if
    $\hat{\boldsymbol{\theta}}_a \leftarrow \mathbf{A}_a^{-1}\mathbf{b}_a$
    $p_{t,a} \leftarrow \hat{\boldsymbol{\theta}}_a^\top \mathbf{x}_{t,a} + \alpha\sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1}\mathbf{x}_{t,a}}$
  end for
  Choose arm $a_t = \arg\max_{a\in\mathcal{A}_t} p_{t,a}$ with ties broken arbitrarily, and observe a real-valued payoff $r_t$
  $\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + \mathbf{x}_{t,a_t}\mathbf{x}_{t,a_t}^\top$
  $\mathbf{b}_{a_t} \leftarrow \mathbf{b}_{a_t} + r_t\mathbf{x}_{t,a_t}$
end for
```

Figure 2: Definition of the disjoint linear UCB algorithm taken from paper

Run the LinUCB algorithm with the following command:

```
$ python run.py --model linucb
```

You should see the total_fraction_correct to be above 0.64, though the results may vary per run.

Note 1: please feel free to adjust the –alpha argument, but you don't have to.

Note 2: for arbitrary tie breaking in action selection please simply use `np.argmax()`.

(c) **[5 points (Coding)]**

Is the upper confidence bound making a difference? Please implement the e-Greedy algorithm in `submission.py`. Does eGreedy perform better or worse than Upper Confidence bound? (You do not need to include your answers here).

Run the ϵ-greedy LinUCB with the following command:

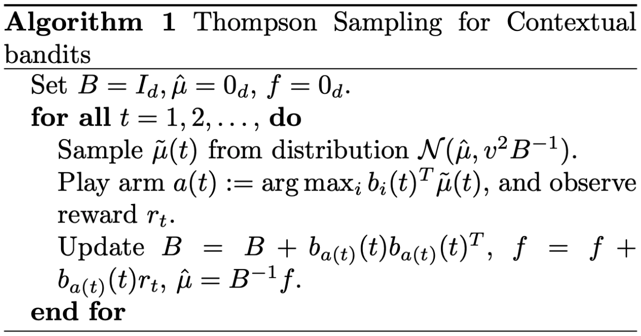
```
$ python run.py --model egreedy
```

You should see the total_fraction_correct to be above 0.61, though the results may vary per run.

Note: please feel free to adjust the –ep argument, but you don't have to.

(d) **[10 points (Coding)]**

Please implement the Thompson Sampling for Contextual Bandits from paper in `submission.py`. Below we have provided a snippet of Algorithm 1 from section 2.2 of the listed paper. For a thorough description of terms please read through the paper.



Algorithm 1 Thompson Sampling for Contextual bandits

```
Set $B = I_d, \hat{\mu} = 0_d$, $f = 0_d$.
for all $t = 1, 2, \ldots,$ do
    Sample $\tilde{\mu}(t)$ from distribution $\mathcal{N}(\hat{\mu}, v^2 B^{-1})$.
    Play arm $a(t) := \arg\max_i b_i(t)^T \tilde{\mu}(t)$, and observe reward $r_t$.
    Update $B = B + b_{a(t)}(t) b_{a(t)}(t)^T$, $f = f + b_{a(t)}(t) r_t$, $\hat{\mu} = B^{-1} f$.
end for
```

Figure 3: Definition of the Thompson sampling algorithm taken from paper

Run the Thompson Sampling algorithm with the following command:

```
$ python run.py --model thompson
```

You should see the total_fraction_correct to be **around** 0.64, though the results may vary per run.

Note: please feel free to adjust the –v2 argument, but you don't have to. This is actually v squared from the paper)

Results

At this point, you should see a plot in your results folder titled `fraction_incorrect.png`. If not, run the following command to generate the plot:

```
$ python run.py
```

This will only work given you have populated csv files for each model in the results folder (e.g. `Fixed.csv`). If you are missing a model's csv file please run the given model once more (as we have outlined in previous questions) to regenerate a csv.

You can expect your results to resemble the plot on the following page.

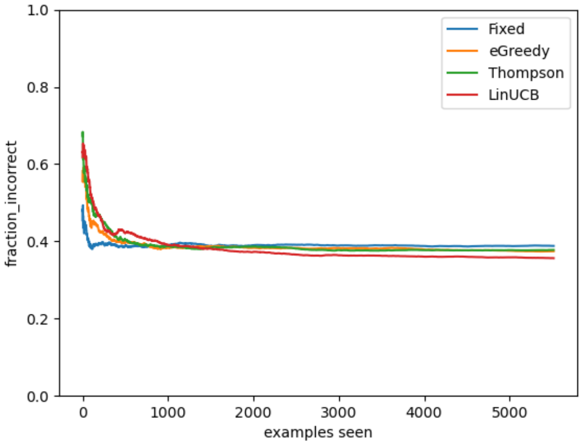


Figure 4: Sample results for the correct implementation of all models.

2 RL for personalized recommendations

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 ([6]), Spotify ([8]), Yahoo ([7]) and Microsoft ([9]) 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 ([5]).

Content recommendations take place in a dynamical system containing feedback loops ([1]) 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 ([2]), 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 ([3]).

We have created a (toy) simulation that allows you to model a video recommender system as a contextual bandit problem.[2] 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 above, N_a was fixed: we always chose from three different dosages for all patients. However, video hosting and recommendation sites like YouTube are more 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 `submission.py`. Most of the simulator is implemented for you: the only method you will need to implement is `update_arms()`.

[2] According to [4], YouTube does not currently use RL for their recommendations, but other video recommendation systems do, as noted above.

Implementing the Bandit Algorithm

(a) **[1 point (Coding)]**

For this assignment we will be using the Disjoint Linear Upper Confidence Bound (LinUCB) algorithm from [7]. The hints provided in `submission.py` should help you with this. You have already implemented this in the previous problem but you will have to now fill the code in the `add_arm_params()` method to account for the fact that the number of arms could keep increasing now.

Implementing the arm update strategies

We will now implement three different strategies to add arms to our simulator. Each arm is associated with its true feature vector θ^*_a. This is the d-dimensional feature vector we assigned to each arm when we initialized the simulator. This is the θ 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 θ^*_1 and θ^*_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 θ^*_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:

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

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

$$\theta_a \leftarrow \theta_a + \eta\frac{\partial L}{\partial \theta} \tag{2}$$

Here η 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

$$\theta_a \leftarrow \theta_a + \eta\sum_{i=1}^{K}(\theta_a^T x_i - \theta_i^T x_i)x_i \tag{3}$$

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.

(b) **[7 points (Coding)]** Implement these three methods in the `update_arms()` function in `submission.py`. This should be about 25 lines of code. Hints have been provided in the form of comments.

Analysis

(c) 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**.

(i) **[3 points (Written)]** 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 run.py -t simulator -s 0 1 2 3 4 -u none
```

(ii) **[2 points (Written)]** 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 run.py -t simulator -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?

(iii) **[2 points (Written)]** 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 run.py -t simulator -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?

(iv) **[4 points (Written)]** 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:

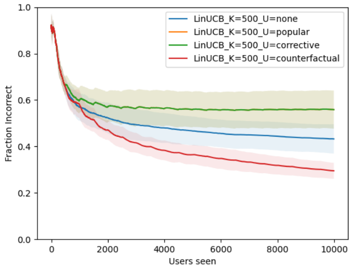
```
        python run.py -t simulator -s 0 1 2 3 4 -u counterfactual -k 500 1000 2500 5000
```

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?

(v) **[4 points (Written)]** 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 run.py -t simulator -s 0 1 2 3 4 -u none popular corrective counterfactual
    -k 500 --plot-u
```

Your plot should look like the following image below:

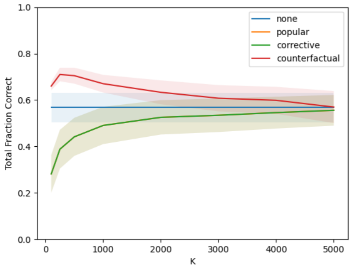


How do the different strategies perform for $K = 500$? Feel free to add your generated plot to the answer!

(vi) **[4 points (Written)]** 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 run.py -t simulator -s 0 1 2 3 4 -u none popular corrective counterfactual
-k 100 250 500 1000 2000 3000 4000 5000 --plot-k
```

Your plot should look like the following image below:



Which strategies get better as you increase K and which strategies get worse? Why do you think this is the case?

Discussion

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

(1) **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.

(2) 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.

(3) 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.

(4) 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".

(d) [**4 points (Written)**] 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)

(e) [**4 points (Written)**] 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.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the —README.md— for this assignment includes instructions to regenerate this handout with your typeset LATEX solutions.

2.ci

2.cii

2.ciii

2.civ

2.cv

2.cvi

2.d

2.e