XCS234 Assignment 4

Due Sunday, February 5 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.

Online Submission: Some questions in this assignment require responses to be submitted interactively within a Gradescope online assessment. For these questions, you should consult your Gradescope dashboard for the availability of this assessment.

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 `src/submission.py`, 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 that the hidden evaluation tests will be able to execute.
- `hidden`: These unit tests are the evaluated 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. These evaluative 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.

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
```

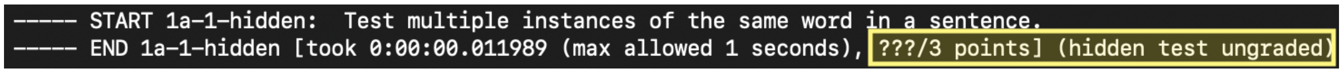
Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

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 - Hidden Tests

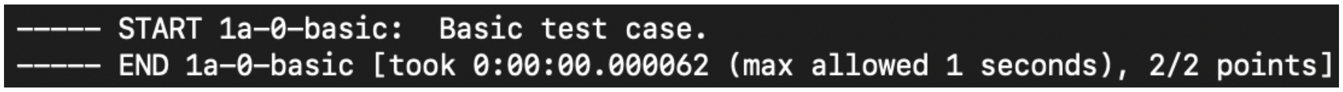
All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:



```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

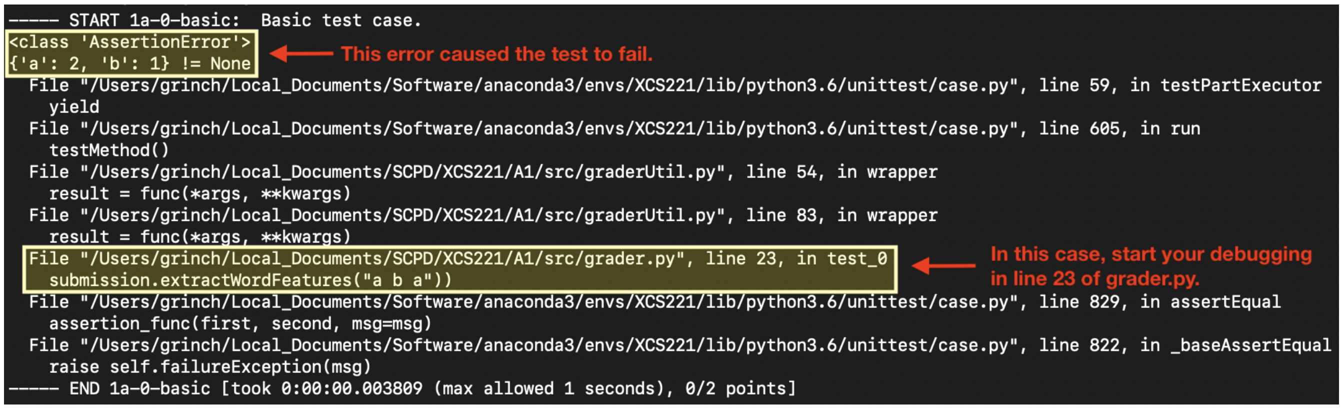
Local Execution - Basic Tests

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:



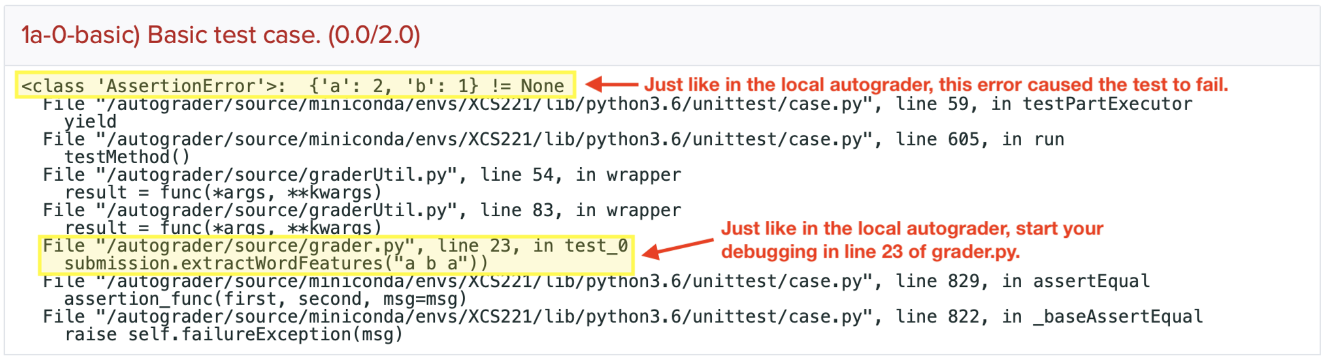
```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", 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. 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.

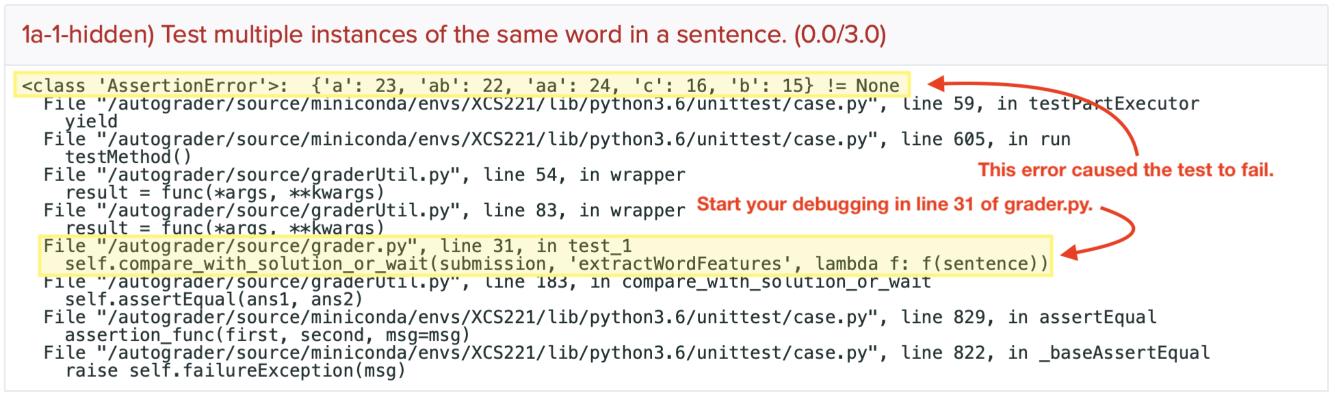


1a-0-basic) Basic test case. (0.0/2.0)

```
<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.



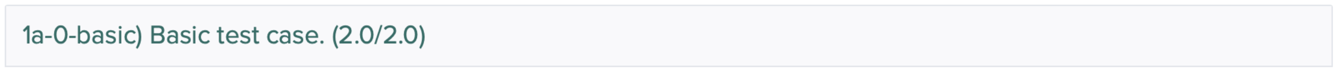
1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<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)
```

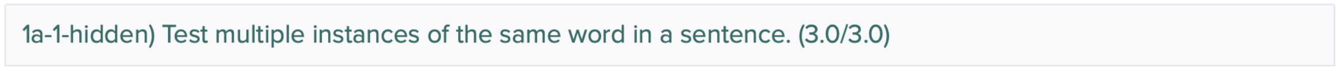
This error caused the test to fail.

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 investigate a Bayesian regret bound for Thompson sampling through a number of derivations.

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`

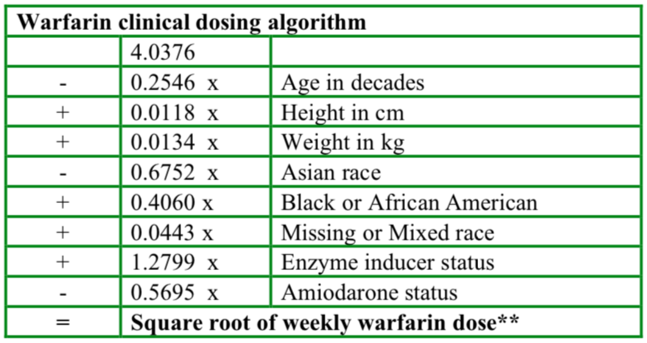
[1]You cannot use `Therapeutic Dose of Warfarin` data as an input to your algorithm.

(a) **[8 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:

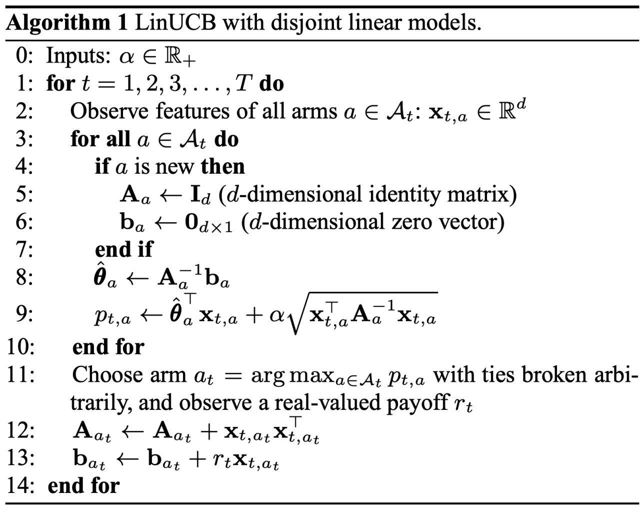
```
$ python run.py --model fixed
$ python run.py --model clinical
```

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) **[19 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.

```
0:  Inputs: $\alpha \in \mathbb{R}_+$
1:  for $t = 1, 2, 3, \dots, T$ do
2:      Observe features of all arms $a \in \mathcal{A}_t$: $\mathbf{x}_{t,a} \in \mathbb{R}^d$
3:      for all $a \in \mathcal{A}_t$ do
4:          if $a$ is new then
5:              $\mathbf{A}_a \leftarrow \mathbf{I}_d$ ($d$-dimensional identity matrix)
6:              $\mathbf{b}_a \leftarrow \mathbf{0}_{d\times 1}$ ($d$-dimensional zero vector)
7:          end if
8:          $\hat{\boldsymbol{\theta}}_a \leftarrow \mathbf{A}_a^{-1}\mathbf{b}_a$
9:          $p_{t,a} \leftarrow \hat{\boldsymbol{\theta}}_a^\top \mathbf{x}_{t,a} + \alpha\sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1}\mathbf{x}_{t,a}}$
10:     end for
11:     Choose arm $a_t = \arg\max_{a\in\mathcal{A}_t} p_{t,a}$ with ties broken arbitrarily, and observe a real-valued payoff $r_t$
12:     $\mathbf{A}_{a_t} \leftarrow \mathbf{A}_{a_t} + \mathbf{x}_{t,a_t}\mathbf{x}_{t,a_t}^\top$
13:     $\mathbf{b}_{a_t} \leftarrow \mathbf{b}_{a_t} + r_t\mathbf{x}_{t,a_t}$
14: 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) **[10 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) **[20 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.

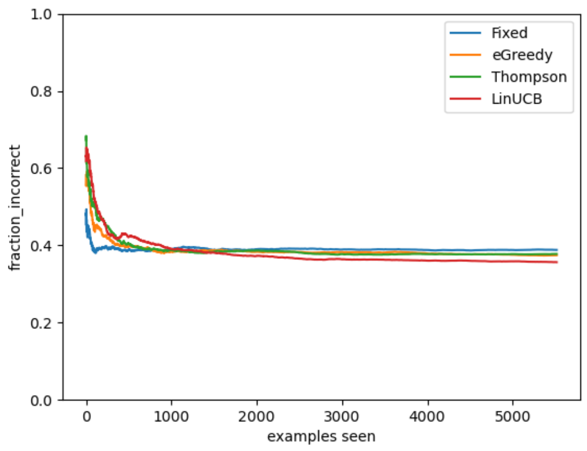


Figure 4: Sample results for the correct implementation of all models.

2 A Bayesian Regret Bound for Thompson Sampling

Consider the K-armed bandit problem: there are K "arms" (actions), and we will choose one arm $a_t \in [K]$ to pull at each time $t \in [T]$, then receive a random reward $r_t \sim p(r \mid \theta, a = a_t)$. Here θ is a random variable that parameterizes the reward distribution. Its "true" value is unknown to us, but we can make probabilistic inferences about it by combining prior belief with observed reward data. We denote the expected reward for arm a (for a fixed θ) as $\mu_\theta(a) := \mathbb{E}[r \mid \theta, a]$.

A *policy* specifies a distribution over the next arm to pull, given the observed history of interactions $H_t = (a_1, r_1, \ldots, a_{t-1}, r_{t-1})$.[2] Formally, a policy is a collection of maps $\pi = \{\pi_t : \mathcal{H}_t \to \Delta(\mathcal{A})\}_{t=1}^T$, where $\mathcal{H}_t$ is the space of all possible histories at time t and $\Delta(\mathcal{A})$ is the set of probability distributions over $\mathcal{A}$. We denote the probability of arm a under policy π at time t as $\pi_t(a \mid, H_t)$.

For a fixed value of θ, the suboptimality of a policy π can be measured by the *expected regret*:

$$R_{T,\theta}(\pi) = \mathbb{E}_H \left[\sum_{t=1}^T \mu_\theta(a^*) - \mu_\theta(a_t) \mid \theta \right]$$

where the expectation is taken with respect to the arms selected, $a_t \sim \pi_t(a \mid H_t)$, and rewards subsequently observed, $r_t \sim p(r \mid \theta, a = a_t)$. We use H as a shorthand for $H_{T+1} = (a_1, r_1, \ldots, a_T, r_T)$.[3] Note that a^* is random because θ is random, but for a given θ it is fixed and can be computed by $a^* = \arg\max_a \mu_\theta(a)$. (Assume for simplicity that there is one optimal action for any given θ.)

Our goal in this problem is to prove a bound on the *Bayesian regret*, which is the expected regret averaged over a prior distribution on θ:

$$\mathrm{BR}_T(\pi) = \mathbb{E}_\theta[R_{T,\theta}(\pi)]$$

We will analyze the *Thompson sampling* (or *posterior sampling*) algorithm, which operates by sampling from the posterior distribution of the optimal action a^* given H_t:

$$\pi_t^{TS}(a|H_t) = p(a^* = a|H_t)$$

We can sample from π_t^{TS} by first sampling $\theta_t \sim p(\theta \mid H_t)$ and then computing $a_t = \arg\max_a \mu_{\theta_t}(a)$.

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

Let $\{L_t : \mathcal{A} \to \mathbb{R}\}_{t=1}^T$ and $\{U_t : \mathcal{A} \to \mathbb{R}\}_{t=1}^T$ be lower and upper confidence bound[4] sequences (respectively), where each L_t and U_t depends on H_t.

Show that the Bayesian regret for Thompson sampling can be decomposed as:

$$\mathrm{BR}_T(\pi^{TS}) = \mathbb{E}_{\theta,H} \left[\sum_{t=1}^T [U_t(a_t) - L_t(a_t)] + [L_t(a_t) - \mu_\theta(a_t)] + [\mu_\theta(a^*) - U_t(a^*)] \right]$$

Hint 1: this equality does not hold in general, its proof requires the use of the property for $\pi_t^{TS}(a \mid H_t)$ listed above.

(b) **[3 points (Written)]**

Now assume the rewards r_t are bounded in $[0, 1]$, $L_t \le U_t$, $L_t \in [0, 1]$ and $U_t \in [0, 1]$. Show that:

$$\mathrm{BR}_T(\pi^{TS}) \le \mathbb{E}_{\theta,H} \left[\left(\sum_{t=1}^T [U_t(a_t) - L_t(a_t)] \right) + T \sum_a \mathbb{I} \left\{ \bigcup_{t=1}^T \{\mu_\theta(a) \notin [L_t(a), U_t(a)]\} \right\} \right]$$

where the notation $\mathbb{I}\{\cdot\}$ refers to an indicator random variable which equals 1 if the expression inside the brackets is true and equals 0 otherwise.

[2]Note: we take history to mean that which is known at the beginning of step t, rather than at the end of step t, so it only goes up to a_{t-1}, r_{t-1}.

[3]The regret does not actually depend on r_T.

[4]In Thompson sampling, the upper confidence bound is not used to select actions; we only introduce it for the purpose of analysis.

(c) **[2 points (Written)]**

Let us now impose a specific form of confidence bounds:

$$L_t(a) = \max\left\{0, \hat{\mu}_t(a) - \sqrt{\frac{2 + 6\log T}{n_t(a)}}\right\}$$
$$U_t(a) = \min\left\{1, \hat{\mu}_t(a) + \sqrt{\frac{2 + 6\log T}{n_t(a)}}\right\}$$

where $\hat{\mu}_t(a)$ is the mean of rewards received playing action a before time t, and $n_t(a)$ is the number of times action a was played before time t. (If action a has never been played at time t, $L_t(a) = 0$ and $U_t(a) = 1$.)

You may take as given the following fact: with L_t and U_t defined as above, it holds that:

$$\forall a, \quad \mathbb{P}_{\theta,H}\left(\bigcup_{t=1}^{T}\{\mu_\theta \notin [L_t(a), U_t(a)]\}\right) \le \frac{1}{T}$$

and thus, the bound from part (b) implies:

$$\mathrm{BR}_T(\pi^{TS}) \le \mathbb{E}_{\theta,H}\left[\sum_{t=1}^{T}[U_t(a_t) - L_t(a_t)]\right] + K$$

To bound the remaining terms, let us use the decomposition $\sum_{t=1}^{T}[U_t(a_t) - L_t(a_t)] = \sum_a \sum_{t\in\mathcal{T}_a}[U_t(a) - L_t(a)]$, where $\mathcal{T}_a = \{t \in [T] : a_t = a\}$.

Show that:

$$\sum_{t\in\mathcal{T}_a}[U_t(a) - L_t(a)] \le 1 + 2\sqrt{2 + 6\log T}\sum_{i=1}^{n_T(a)}\frac{1}{\sqrt{i}}$$

Hint: Consider the possible combinations $U_t(a)$ and $L_t(a)$ and how their difference guarantees the above bounded expression.

(d) **[2 points (Written)]**

Show that:

$$\sum_{i=1}^{n_T(a)}\frac{1}{\sqrt{i}} \le 2\sqrt{n_T(a)}$$

Hint: you can bound the sum by an integral.

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

Use the previous parts to obtain:

$$\mathrm{BR}_T(\pi^{TS}) \le 2K + 4\sqrt{KT(2 + 6\log T)}$$

Hint: you may find the AM-QM inequality $\frac{1}{n}\sum_{i=1}^{n} x_i \le \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$ helpful.

(f) **[2 points (Online)]**

Please refer to questions 1.1 of the Gradescope online assessment `A4 (Quiz)`.

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.a

2.b

2.c

2.d

2.e