Prodigy Algorithmic Design Challenge

Stepan Oskin

December 16, 2019

Abstract

You are given a list of tasks $T = T_1, T_2, \ldots, T_n$. Each task is to be done on a computer. Some tasks are easy, other tasks are hard. (Some examples of tasks could be typing, or coding, or answering multiple-choice questions.) You also have a set of users U who have different abilities on these tasks. The goal of this challenge is to produce a pseudo-code heuristic to solve each step of the challenge, supported by explanations, assumptions, or working code.

Question 1—Estimate the difficulty of each task

1.1 Question 1

You want to estimate the difficulty of each task. Each user is given a random sample of tasks from T to attempt. How can you use the performance of users (on the random task sample they are given) to estimate the difficulty of all the tasks in T? (You can only use observations on the computational tasks - no biometrics, direct observation, etc. Imagine you can see what is on a user's computer screen but nothing else.)

1.2 Solution

1.2.1 Absolute and relative difficulty of a task T_k

Since it is given that users in U have different abilities on tasks in T, when discussing the assessment of the difficulty of a task T_k there are two distinct definitions of difficulty that have to be considered:

1. absolute measure of difficulty for task T_k

   One of the ways to define the absolute measure of difficulty for a given task T_k, or $D_{abs}(T_k)$, is through empirically observed percentage of correct responses from all the answers, $p(T_k)$, that were provided by the subset of users U_{T_k} who attempted the given task T_k. This measure indicates the difficulty of a given task as expressed by the ratio of users who were able to solve it correctly.

2. relative measure of difficulty for task T_k

Relative measure of difficulty for the task T_k in relation with the user U_i, or $D_{rel}(T_k, U_i)$, can be defined as $D_{abs}(T_k)$ corrected for the current level of abilities of a given user U_i. This measure reflects the fact that the two users U_i and U_j might find the same task T_k to have different level of difficulty, depending on their current abilities in a given subject. Furthermore, the same user U_i can find the same task T_k to have higher or lower level of difficulty at different points in time (e.g., if the user U_i improves their skills through practice). $D_{rel}(T_k, U_i)$ is intended to reflect both of these differences when assessing the relative difficulty of a given task T_k in relation to a given user U_i.

This section is concerned with the absolute difficulty, $D_{abs}(T_k)$, of a task as reflected by the performance of the subset of users U_{T_k} who have attempted the task T_k. Relative task difficulty $D_{rel}(T_k, U_i)$ will be discussed further in section 3.2.2.

1.2.2 Determining absolute task difficulty $D_{abs}(T_k)$ using difficulty index

Difficulty index presents a basic approach to determining absolute task difficulty: difficulty p of each task T_k can be determined as the fraction of the correct responses provided by all the users who attempted it:

$$D_{abs}(T_k) = P(Ans = Corr|T_k) = \frac{\text{Correct answers}}{\text{Total answers}} \tag{1.1}$$

This approach assumes that each task in T was only shown to the subset of users U_{T_k}, for whom this task is relevant (e.g., tasks and users are grouped by school year). Also, in this approach, individual skill level of users is not taken into account; instead, absolute task difficulty $D_{abs}(T_k)$ can be interpreted as the expectation of the probability of an average user U_i solving the given task T_k correctly, as reflected by the subset of users U_{T_k} who attempted this task:

$$D_{abs}(T_k) = E[P(Ans = Corr)|T_k, U_i \in U_{T_k}] \tag{1.2}$$

1.2.3 Uncertainty due to the size of subsets U_{T_k}

Since in our case each task T_k is only attempted by a subset of users U_{T_k}, when using the definition of absolute task difficulty $D_{abs}(T_k)$ presented in equation 1.2, the assumption is that the expectation of probability of correct answer for an average user from the sample U_{T_k} is close to that for an average user from the whole population U:

$$E[P(Ans = Corr|T_k, U_i \in U_{T_k})] \approx E[P(Ans = Corr|T_k, U_i \in U)] \quad (1.3)$$

Since it is said that users are given randomized subsets of tasks from T, given the large number of users and tasks, it can be assumed that for most tasks in T, each task T_k is shown to a large random sample of users U_{T_k} whose skill level and platform usage patterns are representative of the whole population of users in U; in this case, absolute difficulty $D_{abs}(T_k)$ of the task T_k can be assumed to offer an unbiased estimate of absolute task difficulty with respect to all users in U.

However, this assumption is more likely to hold true in cases where most subsets of users U_{T_k} are sufficiently large, that is, enough users have attempted to solve most tasks in T. To determine the uncertainty of this method, distribution of sample sizes U_{T_k} can be compared with the size of the population U for all tasks in T. The effect of sample sizes on difficulty estimates can be mitigating by using the uncertainty of each estimate as a weight coefficient when combining empirical absolute difficulty $D_{abs}(T_k)$ with expert estimate $D_{expt}(T_k)$ to determine the composite absolute task difficulty $D_{comp}(T_k)$, which will be discussed in section 2.2.2. In addition, analysis of the distribution of D_{abs} obtained empirically could be used to test the assumption and inform the design of a more meaningful scoring system that can be used to determine the absolute difficulty of a task.

1.2.4 Pseudocode

```
for each task $T_i$ in $T$:
    $D_{abs}(T_k) = \frac{\text{Correct answers}(T_i)}{\text{Total answers}(T_i)}$
```

Question 2—Incorporate expert rating of difficulty of each task

2.1 Question 2

Now suppose you are given a set L of task lists $L_1, L_2, \ldots, L_n$. Each task list contains some tasks in estimated ascending difficulty order as measured by some integer D, where $D = 0$ for an extremely trivial task and $D = 50$ for the hardest task imaginable. However, these given task difficulties are only estimates by experts - in practice each user who tries a task might find that task easier or harder than the expert thought it would be. How would you use the given set of task lists along with the actual performance of users on each task to generate a single combined task list with better estimated difficulties for all tasks? (Please assume that for any task T_i, the difficulty estimate D_i can vary depending on the specific expert's evaluation of difficulty for T_i - that is to say, experts will likely disagree on how difficult any given task is. You can also assume that the same task may appear on multiple task lists.)

2.2 Solution

2.2.1 Combining individual expert assessments into a single expert difficulty estimate $D_{expt}(T_k)$

First, it is necessary to obtain the combined expert difficulty estimates $D_{expt}(T_k)$ for each task T_k in T by combining all the ratings $D_i(T_k)$ found for a given task on different task lists in L. Since no specific weight is given to individual expert opinions, the combined expert difficulty estimate

$D_{expt}(T_k)$ can be taken for each task T_k as the mean of all estimates $D_i(T_k)$ from every task list in L which contains the task T_k:

$$D_{expt}(T_k) = \left\lfloor \frac{1}{m} \sum_{i=1}^{m} D_i(T_k) \right\rfloor \tag{2.1}$$

where m is the count of task lists in L which include the task T_k.

In the equation 2.1, $D_{expt}(T_k)$ can be left as a real number and be directly converted to a probability of a correct answer using the equation 2.3 that will be introduced in the following section. However, the preferable way of converting the 0–50 scale used by experts to mark D_{expt} to a 0–1 scale comparable with probabilities expressed by D_{abs} involves grouping questions by D_{expt}(to be discussed in the following section), and thus works better for a discrete set of values. For this purpose in equation 2.1, the mean of all expert estimates is rounded down to the nearest integer. This process introduces a pessimistic bias by rounding the mean difficulty values down, but since the task difficulty scale used by experts ranges from 0 to 50, this effect is assumed to be mild.

2.2.2 Combining empirically determined absolute task difficulty $D_{abs}(T_k)$ with expert estimates $D_{expt}(T_k)$

In order to obtain the composite absolute difficulty measure $D_{comp}(T_k)$ of a task T_k, the two independently obtained measures of task difficulty $D_{abs}(T_k)$ and $D_{expt}(T_k)$ need to be combined into a single measure; to facilitate this, they need to be converted to a comparable scale. As was discussed in section 1.2.2, $D_{abs}(T_k)$ represents the empirically obtained expected value of the probability that an average user in U can solve the task T_k (as observed by performance of users in the subset U_{T_k}). In contrast, expert task difficulty estimate $D(T_k)$ is assigned using an integer scale from 0 to 50, with 0 corresponding to an extremely trivial task and 50 representing the hardest task imaginable. Since $D_{expt}(T_k)$ was defined in section 2.2.1 as the rounded down mean of all D_i for a given task T_k, it will also fall onto an integer scale from 0 to 50.

To bring the two metrics onto a single composite absolute difficulty measure, $D_{comp}(T_k)$, the scale of D_{expt} must be mapped onto the corresponding probabilities D_{expt_prob} of the correct answer by an average user in U, similar to the ones that were estimated empirically for $D_{abs}(T_k)$ in section 1.2.2:

$$D_{expt\_prob}(T_k|D_{expt}(T_k) = D_i) = E[P(Ans = Corr|D_{expt}(T_k) = D_i)] \tag{2.2}$$

The most simple way of converting the 0–50 difficulty scale used by experts to a 0–1 probability scale is through linear interpolation. $D(T_k) = 0$ can be assumed to correspond to difficulty $D_{abs}(T_k) = 1.00$ (all users can solve the most trivial task), and $D(T_k) = 50$ to correspond to $D_{abs}(T_k) = 0.0$ (no user can solve the hardest task imaginable). This linear relationship between expert difficulty estimate D_{expt} and the absolute probability of the correct answer $D_{abs} = p$ can be represented as follows:

$$D_{expt\_prob}(T_k|D_{expt}(T_k) = D_i) = 1 - 0.02 \cdot D_{expt} \tag{2.3}$$

where D_i is an integer value from 0 to 50.

However, a better approach to converting expert difficulty estimates $D_{expt}(T_k)$ into probabilities $D_{expt\_prob}(T_k)$ is to calibrate expert task difficulty estimates to actual performance of users in U. Doing this empirically would require mapping each discrete value of D to the corresponding ratio of correct answers to all attempts for all the tasks in T with the difficulty level of D_i; this will allow to capture the non-linear relationship between expert difficulty estimates and performance of users in U. This process could be repeated in order to provide an updated calibration of static expert difficulty estimates to evolving user population.

After both task difficulty measures $D_{abs}(T_k)$ and $D_{expt}(T_k)$ have been brought to the same scale (expected probabilities of the correct answer given by a user from U), the final composite task difficulty measure can be defined as the mean of the two difficulty measures:

$$D_{comp}(T_k) = \frac{D_{abs}(T_k) + D_{expt\_prob}(T_k|D_{expt}(T_k) = D_i)}{2} \tag{2.4}$$

Alternatively, if a given preference exists between the two difficulty measures, D_{comp} can be defined as the weighted sum of the two. For example, the uncertainty of the $D_{abs}(T_k)$ arising from the limited size of the sample of users U_{T_k} who attempted the task T_k (discussed in section 1.2.3) can be used as a weight coefficient; while the number of users who tried solving T_k remains small, more weight can be given to expert difficulty estimate; as the number of users in U_{T_k} grows, the role of expert assessment diminishes with more reliance being put on empirical data.

2.2.3 Converting absolute composite task difficulty $D_{comp}(T_k)$ into a discrete random variable $D_{cat}(T_k)$

The main task of the adaptive algorithm for designing custom task lists for each user session (to be discussed in the following sections) is to determine the sequence of task difficulties to be presented to a user. Since for the purposes of the implementation of a planning agent it is more convenient to work with discrete action spaces rather than with continuous ones, question difficulty can be categorized by binning the values of D_{comp} into a discrete categorical absolute difficulty D_{cat} using a rough "rule-of-thumb" classification according to the difficulty index, as is shown in table 2.1.

Task difficulty ($D_{cat}(T_k)$)	$D_{comp}(T_k)$	% correct
Very trivial	$p > 0.8$	Over 80%
Trivial	$0.6 < p \leq 0.8$	From 60% to 80%
Moderate	$0.4 < p \leq 0.6$	From 40% to 60%
Hard	$0.2 < p <\leq 0.4$	From 20% to 40%
Very hard	$p \leq 0.2$	Less than 20%

Table 2.1: Categorizing questions by absolute difficulty using difficulty index.

So far, all the consideration has been given to rating absolute task difficulty, as defined by empirical user performance or expert judgement. However as was discussed in section 1.2.1, due to the difference in ability levels shown by users in U on the tasks in T, there are two definitions of task difficulty that need to be considered for the purposes of custom task list design: absolute task difficulty for a given task T_k and relative task difficulty for a given pair of task T_k and user U_i. In the following section, the absolute categorical task difficulty $D_{cat}(T_k)$ that was produced from the composite absolute task difficulty index $D_{comp}(T_k)$ will be used in combination with the user ranking by experience to produce relative task difficulty, $D_{rel}(T_k, U_i)$.

2.2.4 Pseudocode

```
# combine expert estimates into a single absolute difficulty assessment
  for each task $T_k$ in $T$:

     Initialize empty list of ratings
     for each task list in $L$ containing $T_k$:
       append rating $D_i(T_k)$ to the list of ratings
     $D_{expt}(T_k) = \left\lfloor \frac{1}{m} \sum_{i=1}^{m} D_i(T_k) \right\rfloor$, where $m$ in the number of task lists in
$L$ containing task $T_k$
```

# convert from D_{expt} 0–50 scale to D_{expt_prob} 0–1 scale using linear relationship

# Note: preferred conversion method involves calibration of D_{expt_prob} estimates using empirical data

$$D_{expt\_prob}(T_k|D_{expt}(T_k) = D_i) = 1 - 0.02 \cdot D_{expt}$$

# determine the combined absolute task difficulty

$$D_{comp}(T_k) = \frac{D_{abs}(T_k) + D_{expt\_prob}(T_k|D_{expt}(T_k) = D_i)}{2}$$

Question 3—Adaptive algorithm for building custom task lists

3.1 Question 3

Based on your answer for 2 (i.e., given a single combined task list with estimated difficulties), suppose you wanted to build an adaptive algorithm to order the sequence of tasks each user is given. You want to create a custom list of tasks for each user. This custom list should give tasks of a suitable level of difficulty for each user, and you want the difficulty of tasks each user does to gradually increase over time. How would you approach this?

3.2 Solution

3.2.1 User ranking based on experience

Users in U are said to have different skill levels on tasks in T; in addition, their abilities change with time, which also needs to be taken into account when building custom task lists for subsequent sessions. Therefore, in order to make informed selection of tasks to be shown to the user U_i in a given session, the algorithm needs to take into account the current level of abilities of the user U_i at the time of the session. Since all the tasks in T have previously been assigned an absolute difficulty category D_{cat} (introduced in section 2.2.3) based on the composite absolute task difficulty measure D_{comp} (introduced in section 2.2.2), a user ranking system can be established based on the history of each user in U, including the number and absolute difficulty of all tasks solved over all of their previous sessions.

Since all the users are given random samples of tasks from T, distribution of absolute difficulty of attempted tasks may vary from user to user. In addition, users may have different number of previous sessions, with some of the users having attempted significantly more tasks than others. Thus, the user ranking system needs to account for both the variance in the number of questions solved correctly by a user U_i, and for variation in their absolute difficulty $D_{cat}(T_k)$. The total number of tasks attempted (including the ones that were solved incorrectly) is assumed to not contribute much to the true level of user's ability.

To rank users in U based on their current ability, a new experience metric, $Ex(U_i)$, can be established to account for the number and difficulty of tasks previously solved by each user U_i over all of their past sessions. This metric intends to represent relative skill level of the user U_i compared to other users in U, as shown by their past performance on tasks in T. The experience metric is computed as a weighted sum of all tasks solved by a user corrected by their experience coefficients, which are assigned based on the absolute difficulty of each task. An example of the experience weight coefficients Ec related to absolute categorical task difficulty D_{cat} is shown in the table 3.1.

Absolute task difficulty D_{cat}	Experience coefficient $Ec(D_{cat})$
very trivial	0.1
trivial	0.5
moderate	1.0
hard	5.0
very hard	10.0

Table 3.1: Experience coefficients $Ec(D_{cat})$ assigned to each task difficulty category D_{cat} for the purpose of ranking users by relative skill level, as shown by their past performance on tasks in T. The order of magnitude difference in Ec coefficients intends to represent the assumption that a user who can solve very hard questions with at least 50% probability is likely to solve very trivial questions with almost 100% chance. At the same time, a user who can only solve a large number of very trivial questions is not likely to be able to answer very hard questions at all. Other users fall somewhere in between the two extremes.

Thus, experience of a given user $Ex(U_i)$ can be expressed as:

$$Ex(U_i) = 0.1 \cdot very\ trivial + 0.5 \cdot trivial + 1.0 \cdot moderate + \\ +5.0 \cdot hard + 10.0 \cdot very\ hard \tag{3.1}$$

where *very trivial*, *trivial*, *moderate*, *hard* and *very hard* are the counts of tasks of absolute difficulty level D_{cat} successfully completed by a given user U_i. After calculating the experience $Ex(U_i)$ for each user in U, all users can be ranked by Ex and split into groups, thus reducing a continuous random variable $Ex(U_i)$ to a discrete user category GU_j.

For the purposes of this challenge, the experience coefficients assigned to each task difficulty in equation 3.1 (also shown in table 3.1) have been chosen arbitrarily. However, their intention is to produce a ranking system for all users in U which allows the users to be stratified by their ability, ideally splitting them into distinct groups according to their expected probability of correctly solving a task of a given absolute difficulty. The basic idea behind the chosen values was to create an order of magnitude difference in experience coefficients $Ec(D_{cat})$ between different task difficulty categories, thus stratifying the users who can solve questions of increasing difficulty.

Absolute task difficulty	Probability of the correct answer $E[P(Ans = Corr \mid D_{cat}(T_k), U_i \in GU_j)]$				
$D_{cat}(T_k)$	GU1	GU2	GU3	GU4	GU5
Very trivial	0.5	0.75	0.9	0.95	0.99
Trivial	0.25	0.5	0.75	0.9	0.95
Moderate	0.1	0.25	0.5	0.75	0.9
Hard	0.05	0.1	0.25	0.5	0.75
Very hard	0.01	0.05	0.1	0.25	0.5

Table 3.2: Intended outcome of the user ranking system based on the new experience metric Ex. GU_j represent categories of users produced by ranking users in U based on their experience Ex and grouping them using the natural breaks identified in the distribution of Ex values. Each group is expected to show some variance in terms of success rates on certain task difficulties, but it is assumed that given a large enough population of users in U, each group will represent a distinct sub-population of users with a certain skill level, as expressed by their performance on tasks in T. $E[P(Ans = Corr \mid D_{cat}(T_k), U_i \in GU_j)]$ represents the expectation of probability that a user U_i from the group GU_j will be able to solve the given task T_k of absolute difficulty level $D_{cat}(T_k)$.

Calibration of experience coefficients used in equation 3.1 can be performed using empirical data, aiming to obtain user stratification analogous to the one presented in table 3.2. In addition, the number of recognized user categories can be expanded or reduced (example in table 3.2 is presented for five groups that could be produced using a technique for identifying natural breaks in the distribution of user experience Ex, such as the Fischer-Jenks optimization algorithm). Experience coefficients Ec should be selected to produce a distribution of experience Ex for all users in U with identifiable natural breaks which correspond to more or less homogeneous groups of users as reflected by their mean probability (and its variance) to solve the tasks of a given absolute difficulty D_{cat}; table 3.2 presents an example of how such split could look like.

To sum up, the process of user ranking outlined above results first in the introduction of a new continuous random variable Ex for each user in U representing user skill level, as indicated by their past performance on tasks in T. User experience $Ex(U_i)$ is computed as a weighted sum of the number of tasks previously solved by the user U_i; tasks are counted by categorical absolute difficulty $D_{cat}(T_k)$, counts are weighted using predefined experience coefficients Ec for each task difficulty. Then, to simplify the logic of the task list-building agent, user experience rankings are converted to discrete user groups (for example, users can be categorized by identifying the natural breaks in the distribution of Ex using Fisher-Jenks optimization algorithm). Each group is intended to represent a subpopulation of users with distinct expected probability (as expressed by group mean and variance) to solve a task of a given absolute difficulty category $E[P(Ans = Corr|D_{cat}(T_k), U_i \in GU_j)]$.

The process of re-ranking all the users in U can be repeated at fixed time intervals to recognize the changes in user experience as more tasks are being solved; groups GU can be re-assigned regularly to users based on the most up-to-date history of each user's performance. Ultimately, the user category GU_j, along with the categorical absolute task difficulty D_{cat}, are used by the task list-building agent to design the custom task list to be shown to the user U_i in a given session. The possible types of logic for the task list-building agent are discussed in the remainder of this document.

3.2.2 Relative task difficulty $D_{rel}(T_k, U_i)$

Since it is said that users in U have different abilities on tasks in T, when discussing the difficulty of a task T_k there are two distinct definitions of difficulty that have to be considered: absolute and relative task difficulty.

Absolute categorical task difficulty $D_{cat}(T_k)$ was introduced in section 2.2.3. After user categories (introduced in section 3.2.1) have been determined, relative difficulty $D_{rel}(T_k, U_i) = D_{rel}(D_{cat}(T_k), U_i \in GU_j)$ of each task and user pair can be established by matching user and task categories using table 3.3. The matching is done according to the probability estimates of correct answers for each user group and task difficulty category that was outlined in table 3.2.

Absolute task difficulty	Relative task difficulty $D_{rel}(D_{cat}(T_k), U_i \in GU_j)$				
$D_{cat}(T_k)$	GU1	GU2	GU3	GU4	GU5
Very trivial	Average	Easy	Too easy	Too easy	Too easy
Trivial	Hard	Average	Easy	Too easy	Too easy
Moderate	Too hard	Hard	Average	Easy	Too easy
Hard	Too hard	Too hard	Hard	Average	Easy
Very hard	Too hard	Too hard	Too hard	Hard	Average

Table 3.3: Relative task difficulty $D_{rel}(T_k, U_i) = D_{rel}(D_{cat}(T_k), U_i \in GU_j)$. GU_j represent categories of users produced by ranking all users in U based on experience $Ex(U_i)$ and grouping them using the natural breaks identified in the distribution of Ex values. $D_{cat}(T_k)$ represents absolute categorical task difficulty introduced in section 2.2.3. The resultant categories of $D_{rel}(D_{cat}(T_k), U_i \in GU_j)$ represent the expectation of probability that the user $U_i \in GU_j$ can solve the task T_k as was outlined in table 3.2.

3.2.3 Adaptive algorithm to build custom task lists

The goal of the adaptive task list-building algorithm is to generate custom task lists to be shown to user U_i; the tasks need to have a suitable level of difficulty for the user U_i at the time of each session. The categories for relative task difficulty D_{rel} introduced in section 3.2.2 are intended to reduce the size of the state and action spaces for the task list-building agent: three out of five categories of relative task difficulty $D_{rel}(D_{cat}(T_k), U_i \in GU_j)$ shown in table 3.3 form the state space for the adaptive task list-building algorithm.

- once a user U_i, previously classified by past performance to belong to group GU_j, starts a new session, the algorithm needs to select the sequence of tasks from T to be present to the user in the current session.

- at the start of each session, the agent creates three pools with relatively "easy", "average" and "challenging" tasks by randomly choosing tasks from T with the corresponding absolute difficulty level D_{cat}, as indicated in table 3.3. For example, for a user from the group GU_1, an appropriate absolute task difficulty to form a pool of relatively "average" tasks would be "very trivial":

  $D_{cat}(T_k|D_{rel}(T_k, U_i \in UG_1) = Average) = Very\ trivial$

- tasks that are categorized as either "too hard" or "too easy" according to D_{rel} (as seen in table 3.3) are not considered to be included during the current session.

- following the policy π, the task list-building agent selects a sequence of pools to draw tasks from (e.g., easy, easy, challenging, ...,) in the current session. The sequence can either be static (selected by a pre-defined logic), or it can be adaptive based on the correctness of responses given by the user U_i in the current session.

- based on the defined sequence, the custom task list is built by randomly selecting tasks from corresponding pools without replacement, either in batch or after each user response. This way, the agent does not need to deal with the state space of the size of all available tasks for each task choice, and instead, each time it needs to select one from only three available options, each representing the difficulty level of the pool from which the next task should be drawn.

This section discusses several "baseline" policy options for reflex agents using pre-defined hard-coded logic, the solution to question 4 discusses an adaptive planning agent using Markov decision process (MDP) framework that attempts to take into account current frustration level experienced by the user U_i during the ongoing session. Below are some of the examples of the basic hard-coded logic for task list-building algorithm:

- Same difficulty policies, π_{easy}, $\pi_{average}$ or π_{hard}: the most basic policies instruct the agent to always select questions of the same difficulty in a given session. This method is not adaptive and does not offer any flexibility of adjusting the difficulty of tasks shown based on user responses; it could be used if the user would want to manually control the difficulty of tasks to be selected in each session.

- Random policy, π_{random}: another basic policy includes the agent drawing a task from "easy", "average" or "hard" pool at random using uniform distribution.
- Weighted random policy, π_{random_w}: a slight improvement over the random policy π_{random}, assigns weights to each category for random choice (e.g., 50% easy questions, 30% average, 20% hard).
- Hard-then-easy policy, π_{hard_easy}: a slight improvement over weighted random policy π_{random_w}, aims to recognize the growing user fatigue during a given session, and adjusts the weights used for random choice of task difficulty accordingly (higher chance of challenging tasks at the start of a session; as more tasks are shown, chance of picking a task from the easy pool increases).

The advantage of the basic hard-coded methods discussed above is the fact that the custom task list can be built out in advance at the start of the session, with no additional computation required while the session continues. However, these methods do not take into account the correctness of responses given by the user U_i to the tasks that were presented in the current session.

Since incorrect responses to tasks result in in-game penalties (missed attacks, lost battles, etc.), they can contribute to a growing level of frustration experienced by a user during a session; at the same time, users do not learn much by accomplishing only the tasks that are relatively easy for them to solve. Thus, showing tasks of higher relative difficulty to users would result in the improvement of their learning outcomes; at the same time, picking a lot of questions that are qualified as "challenging" for a given user U_i will likely result in a higher rate of wrong responses and higher frustration levels due to in-game losses experienced by a user U_i during a session.

In order to show higher difficulty tasks while avoiding high levels of frustration, an agent could take recent user responses into account when choosing the next task to be shown. Policy π_{adapt_simple} presents an adaptive deterministic logic for difficulty selection for task T_{t+1} that takes into account recent responses given by the user U_i on tasks T_t and T_{t-1}:

- Adaptive deterministic policy π_{adapt_simple}:
  - π_{adapt_simple} starts by picking two questions from the "easy" pool
  - for each two correct responses in a row, the difficulty of the following task T_{t+1} is raised (e.g., after two "easy" tasks answered right in a row, the next will be "average", etc. If two "challenging" questions are solved correctly, next one will also be challenging).

– after each wrong answer, the difficulty of the following task T_{t+1} is dropped (e.g., after an incorrect "average", the next task will be "easy", etc. After an incorrect "easy" task, the next one will be "easy" as well.)

However, the nature of the environment in which the agent operates is stochastic, as the agent can never be absolutely sure if the user will be able to solve any given task T_k; both the absolute $D_{abs}(T_k)$ and the relative $D_{rel}(T_k, U_i)$ task difficulty estimates are probabilistic in their nature. The following section presents an approach to designing the task list-planning agent as a sequential stochastic decision making process through a Markov decision process framework which takes into account the probabilistic nature of the environment.

3.2.4 Pseudocode

```
# identify user categories by ranking the users according to experience $Ex$

    for user $U_i$ in users $U$:
```

$$\begin{aligned} Ex(U_i) = 0.1 \cdot \mathit{very\ trivial} + 0.5 \cdot \mathit{trivial} + 1.0 \cdot \mathit{moderate} + \\ + 5.0 \cdot \mathit{hard} + 10.0 \cdot \mathit{very\ hard} \end{aligned} \tag{3.2}$$

```
    # identify natural breaks in the $Ex$ distribution using Fisher-Jenks optimization algorithm
    $breaks = jenks.natural\_breaks(U[Ex])$
    Assign user categories by binning users by experience within the natural breaks identified in $Ex$ distribution

    # logic for the simple adaptive agent following the policy $\pi_{adapt\_simple}$ (2 right: diff up, 1 wrong: diff down), uses relative difficulty

    # fill task pools with tasks of appropriate difficulty for the current user
    if user in $GU_1$:
        pool_hard = random choice from $T$ where $D_{abs}(T_k) == Trivial$
       ...
        Other pools are filled in a similar fashion based on user category.
```

```
# iteratively select questions to be shown to the user based on their responses
# starting task difficulty
current_diff = ‘easy’
# initialize variables to store the results of previous responses
num_correct = 0

while keep_playing:
    # show the task of selected difficuly to the user and record their response
    response = task.show(difficulty=current_diff)
    if response == correct:
        num_correct += 1
    else:
        num_correct = 0
        if current_diff == Average:
            current_diff = Easy
        if current_diff == Hard:
            current_diff = Average
        # easy difficulty is not adjusted following an incorrect answer
    if num_correct = 2:
        num_correct = 0
        if current_diff == Average:
            current_diff = Hard
        if current_diff == Easy:
            current_diff = Average
        # hard difficulty is not adjusted following an incorrect answer
```

Question 4—Taking current user frustration into account

4.1 Question 4

How would you modify your algorithm from part 3 to estimate a user's frustration level F where $F = 0$ denotes no frustration and $F = 50$ denotes the highest level of frustration a user can safely experience without giving up at each point in time?

4.2 Solution

4.2.1 Estimating user's frustration level F_t

As was discussed in section 3.2.3, incorrect responses given by the user to presented tasks result in in-game penalties (e.g., missed attacks, lost battles, etc.); these events are likely to contribute to a growing level of frustration experienced by a user during a session. At the same time, users do not learn much by accomplishing only the tasks that are relatively easy for them to solve; thus, showing the tasks of higher relative difficulty during a session would result in the improvement of learning outcomes.

When selecting a sequence of tasks to be shown to the user U_i during a session, a trade-off needs to be established between the relative difficulties of tasks to be shown and the risk of causing the user to give too many wrong responses, reach their frustration limit and terminate the session. Several basic policies presented in section 3.2.3 (same difficulty policies π_{easy}, $\pi_{average}$, π_{hard}, or random policies π_{random}, π_{random_w} and π_{hard_easy}) did not take into account the correctness of responses given by the user U_i in the current session, and instead selected the sequence of task difficulties based solely of user ranking. The deterministic adaptive policy π_{hard_easy} selected the difficulty of the next question based on the correctness of one or two past responses, but did not explicitly take into account the probabilistic nature of task difficulty definition. This section introduces an attempt to incorporate the current frustration level F_t experienced by the user U_i during a session, as well as the probability of the user U_i solving a task of a given relative difficulty, into the logic of the task list-building algorithm.

First, the current level of frustration F_t experienced by the user U_i during any state of the session needs to be established. It is given that frustration F needs to be marked on such scale so that 0 represents no frustration and 50 representing the highest level of frustration that the user U_i can safely experience without terminating the session. In order to incorporate user frustration level into a probabilistic framework, such as Markov decision process, it makes sense to associate the values of F with the corresponding probability of the user U_i terminating the session at a given time $P(quit|F)$. Since $F = 0$ is given to represent no frustration, it can be matched to 0% chance of quitting:

$$F = 0 \implies P(quit|F) = 0 \tag{4.1}$$

It is also given that $F = 50$ represents the highest level of frustration a user can safely experience without giving up. However, it is not clear how to interpret which level can be considered safe in probabilistic terms, since there is always some chance that a user can terminate a session on every time step. For the purposes of this challenge, a level of user frustration $F = 50$ was assumed to correspond to a 10% chance of user terminating the session at a given time.

$$F = 50 \implies P(quit|F) = 0.1 \tag{4.2}$$

By extending the scale linearly, a level of user frustration above $F = 500$ is assumed to correspond to a 100% chance of user terminating the session at a given time.

$$F >= 500 \implies P(quit|F) = 1.0 \tag{4.3}$$

Overall, user frustration F_t can be converted to the probability of the user terminating the session $P(quit)$ at a given time step t as follows:

$$P(quit|F) = \begin{cases} 0.002 \cdot F & \text{if } F < 500 \\ 1.0 & \text{if } F >= 500 \end{cases} \tag{4.4}$$

For the purposes of this challenge, it is assumed that the current level of frustration is determined solely by the total number of incorrect answers given during the current session. The frustration threshold (corresponds to $P(quit|U_i) = 1$) for each user U_i can be empirically determined as the mean number of incorrect answers per session for all previous sessions logged by U_i. If the user U_i has logged less than 10 sessions, the frustration threshold from all users in U can be used instead.

$$F(P(quit|F) = 1.0) = \frac{\sum \text{(Count of incorrect answers per session)}}{\text{Number of sessions}} \tag{4.5}$$

4.2.2 Formulating the task difficulty selection problem as a Markov decision process

A Markov decision process is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly uncertain, and partly under the control of a decision maker. MDP framework provides a formalization of the key elements in reinforcement learning (RL), such as value functions and expected reward.

An MDP expresses the problem of sequential decision making, where actions influence next states and the results. In our case, on each time step t the task list-building agent iteratively selects the difficulty level of the task to be shown to the user U_i; selected difficulty influences the learning outcome (main reward r_t) and the probability of the user answering correctly, which in turn also influences the learning outcome as well as the frustration level experienced by user (probability of the user terminating the session at the time step $t + 1$).

An MDP is four-tuple (S, A, P, R) including four key elements:

- S is the state space, with a finite set of states

  In the case of the task list-building agent, agents's state space is a sequence of tasks attempted by the user, with the next task difficulty being selected by the agent on each time step, until the user ultimately terminates the session. After being presented with each new task T_t, the user may react in one of the three possible ways:

$$s_t \rightarrow \begin{cases} \text{quits at the time step } t \\ \text{tries } T_t \text{ and is right} \\ \text{tries } T_t \text{ and is wrong} \end{cases} \tag{4.6}$$

Each state of the agent is defined by:

- the number of tasks previously attempted by the user U_i in the current session (current time step t)
- cumulative reward $G_t = \sum_{i=0}^{t} r_i$, where r_i is the reward (learning outcome) collected at every previous time step
- current frustration level Fc_t experienced by U_i (probability of the user terminating the session at the time step t)

• A is the action space, with a finite set of actions

on each time step t, the agent has three possible pools from which to draw a task for the user U_i: easy, average, or challenging, as defined by the relative task difficulty $D_{rel}(T_k, U_i)$

$$A = \{\text{easy}, \text{average}, \text{challenging}\} \tag{4.7}$$

• P is a transition function, which defines the probability of reaching a state s' from s after performing an action a.

Action	Probability of outcome		
a_t	quits	tries and is wrong	tries and is right
Easy	Fc_t	$0.25 \cdot (1 - Fc_t)$	$0.75 \cdot (1 - Fc_t)$
Average	Fc_t	$0.5 \cdot (1 - Fc_t)$	$0.5 \cdot (1 - Fc_t)$
Challenging	Fc_t	$0.75 \cdot (1 - Fc_t)$	$0.25 \cdot (1 - Fc_t)$

Table 4.1: Transition matrix for the task list-building agent. Fc_t represents current frustration level experienced by the user U_i (probability of the user terminating the session at the time step t). Probabilities of the user answering a given task right or wrong are taken according to the relative task difficulty $D_{rel}(T_k, U_i)$ introduced in section 3.2.2.

The transition function is equal to the conditional probability of reaching a target state s' after performing action a from state s.

$$P(s', s, a) = p(s'|s, a) \tag{4.8}$$

As was discussed above, there are three possible actions a to be taken from each state s (show easy, average or challenging question next) and three possible outcomes s' that may follow (user quits, user tries and is wrong, user tries and is right). Probability of the user quiting at a given time step t is taken as Fc_t (introduced in section 4.2.1); therefore probability of the user attempting the task can be found as $(1 - Fc_t)$ using the rule of compliment. Probability of the user solving the given task can be determined using the relative task difficulty $D_{rel}(T_k, U_i)$ which was introduced in section 3.2.2 (assumed to be 75% of right answers for relatively easy, 50% for average, 25% for challenging).

  Transition matrix for the task list-building agent can be defined using the table 4.1.

- R is the reward function, which determines the value received for transitioning from state s to state s' after performing the action a

  Agent's rewards r_t are determined by the difficulty of the questions selected and by user's ability to answer them correctly (learning outcome). No negative reward is assumed by the user terminating the session; in this case, simply no further gain can be accomplished.

- Markov property

  By definition, the transition function and the reward function posses the Markov property: their values are determined only by the current state, and not from the sequence of the previous states visited. Markov property means that the process is memory-less and the future state depends only on the current one, and not on its history. This way the current state holds all the information. A system with such a property is called fully observable.

  In the current formulation of the task list-building agent, probability of the user quiting is assumed to be determined solely by the current frustration level at time t, probability to answer the questions correctly is determined solely by relative question difficulty, and the reward is determined only by the difficulty of the question and whether the question was answered correctly. Thus, the task list-building agent operates in a fully observable environment, where the current state holds all the information.

- Policy π

  The final objective of an MDP is to find a policy, π, that maximizes the cumulative reward, $\sum_{t=0}^{\infty} R_\pi(s_t, s_{t+1})$, where R_π is the reward obtained at each step by following the policy, π. A solution of an MDP is found when a policy takes the best possible action in each state of the MDP. This policy is known as the optimal policy.

- Return

  When running a policy in an MDP, the sequence of state and action ($S_0, A_0, S_1, A_1, \dots$) is called trajectory or rollout, and is denoted by τ. In each of agents's trajectories, a finite sequence of rewards will be collected as a result of agent's actions. A function of these rewards is called return and in its most simplified version, it is defined as follows:

  $$G(\tau) = r_0 + r_1 + \dots + r_n = \sum_{t=0}^{n} r_t \tag{4.9}$$

  A trivial but useful decomposition of return is defining it recursively in terms of return at time step $t+1$:

  $$G_t = r_t + G_{t+1} \tag{4.10}$$

  The goal of RL is to find an optimal policy, π, that maximizes the expected return as

  $$argmax_{\pi} E_{\pi}[G(\tau)] \tag{4.11}$$

- Value function

  The return $G(\tau)$ provides a good insight into the trajectory's value, but doesn't give any indication of the quality of the single states visited, which can be used by the policy to choose the next best action. The policy has to just choose the action that will result in the next state with the highest quality. The value function does exactly this: it estimates the quality in terms of the expected return from a state following a policy. Formally, the value function is defined as follows:

  $$V_{\pi}(s) = E_{\pi}[G|s_0 = s] = E_{\pi}[\sum_{t=0}^{k} r_t|s_0 = s] \tag{4.12}$$

  The action-value function, similar to the value function, is the expected return from a state but is also conditioned on the first action. It is defined as follows:

$$Q_\pi(s,a) = E_\pi[G|s_0 = s, a_0 = a] = E_\pi[\sum_{t=0}^{k} r_t|s_0 = s, a_0 = a] \quad (4.13)$$

The value function and action-value function are also called the V-function and Q-function respectively, and are strictly correlated with each other since the value function can also be defined in terms of the action-value function:

$$V_\pi(s) = E_\pi[Q_\pi(s,a)] \quad (4.14)$$

Knowing the optimal Q^*, the optimal value function is as follows:

$$V^*(s) = max_a Q^*(s,a) \quad (4.15)$$

because the optimal action is $a^*(s) = argmax_a Q^*(s,a)$.

4.2.3 Modifying the task list-building algorithm to account for frustration

V and Q functions defined above can be estimated by running trajectories that follow the policy, π, and then averaging the values obtained. This technique is effective and is used in many contexts, but can be very expensive considering that the return requires the rewards from the full trajectory. The Bellman equation defines the action-value function and the value function recursively, enabling their estimations from subsequent states. The Bellman equation does that by using the reward obtained in the present state and the value of its successor state. Using the recursive formulation of the return defined in equation 4.10:

$$V_\pi(s) = E_\pi[G_t|s_0 = s] = E_\pi[r_t + G_{t+1}|s_0 = s] = E_\pi[r_t + V(s_{t+1})|s_t = s, a_t \sim \pi(s_t)] \quad (4.16)$$

Similarly, Bellman equation can be adapted to the action-value function:

$$\begin{aligned} Q_\pi(s,a) &= E_\pi[G_t|s_t = s, a_t = a] \\ &= E_\pi[r_t + G_{t+1}|s_t = s, a_t = a] \\ &= E_\pi[r_t + Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a] \end{aligned} \quad (4.17)$$

Dynamic programming (DP) is a general algorithmic paradigm that breaks up a problem into smaller chunks of overlapping subproblems, and then finds the solution to the original problem by combining the solutions of the subproblems. DP can be used in reinforcement learning and is among one of the simplest approaches. It is suited to computing optimal policies by being provided with a perfect model of the environment. DP works with MDPs with a limited number of states and actions as it has to update the value of each state (or action-value), taking into consideration all the other possible states. Since DP algorithms use tables to store value functions, it is called tabular learning.

DP uses bootstrapping, meaning that it improves the estimation value of a state by using the expected value of the following states. DP applies the Bellman equations introduced above in equations 4.16 and 4.17 to estimate V^* and/or Q^*, which can be done as follows:

$$V^*(s) = max_a E[r_t + V^*(s_{t+1})|s_t = s, a_t = a] \tag{4.18}$$

$$Q^*(s,a) = E[r_t + max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}|s_t = s, a_t = a)] \tag{4.19}$$

Then, once the optimal value and action-value function are found, the optimal policy can be found by just taking the actions that maximize the expectation.

An iterative procedure that iteratively improves the value function sequence $V_0, \ldots, V_k$ is called policy evaluation. It uses the state value transition of the model, the expectation of the next state, and the immediate reward. Policy evaluation procedure creates a sequence of improving value function using the Bellman equation:

$$V_{k+1}(s) = E_\pi[r_t + V_k(s_{t+1})|s_t = s] = \sum_a \pi(s,a) \sum_{s',r} p(s'|s,a)[r + V_k(s')] \tag{4.20}$$

Once the value functions are improved, it can be used to find a better policy, which is called policy improvement; it can be done as follows:

$$\pi' = argmax_a Q_\pi(s,a) = argmax_a \sum_{s',r} p(s'|s,a)[r + V_\pi(s')] \tag{4.21}$$

It creates a policy, π', from the value function, V_π, of the original policy, π. The combination of policy evaluation and policy improvement gives rise to two algorithms to compute the optimal policy. One is called policy iteration and the other is called value iteration. Both use policy evaluation to monotonically improve the value function and policy improvement to estimate the new policy. The only difference is that policy iteration executes the two phases cyclically, while value iteration combines them in a single update.

4.2.4 Psudocode

The pseudocode for the policy iteration algorithm introduced above is as follows:

```
Initialize $V_\pi(s)$ and $pi(s)$ for every state $s$

while $\pi$ is not stable:

  # policy evaluation
  while $V_\pi$ is not stable:
    for each state $s$:
      $V_\pi(s) = \sum_{s',r} p(s'|s, \pi(a))[r + V_\pi(s')]$

  # policy improvement
  for each state $s$:
    $\pi = argmax_a \sum_{s',r} p(s'|s, a)[r + V_\pi(s')]$
```