

Supplementary/Complementary Material for STEP

Introduction

In this supplementary material, we discuss, in detail, a step-by-step implementation of our STEP framework. We first cover the various types of RoboTutor data that has been used for our experiment and then, with the help of an example, delve into the details of the three parts of STEP: **S**imulating students, **T**raining a policy, **E**valuating Policies. We have a closer look at how STEP has been implemented in the context of RoboTutor. We then discuss the limitations and future work before concluding.

RoboTutor Data

In this section, we look at the various types of RoboTutor data, namely: **CTA Table**, **Code Drop 2 Matrices Table**, and **Transactions Table**. Since the data was logged from 28 different villages (114-141) and our experiments use only a subset of logged data, we will limit our discussions to only this data: data logged from village 130 containing 42,010 attempts across 8 students.

Activities and KC's

RoboTutor contains 1710 activities in total. When a student is tested, they are usually tested across 10 items under an activity. Each item may be one or more steps, but is usually a single-step item. The user is tested across 22 skills or Knowledge Components (KC's):

1. **Writing Letters**
2. **Writing a Word**
3. **Writing a Sentence**
4. **Write a Number**
5. **Letters**
6. **Syllable Sound**
7. **Familiar Word**
8. **Invented Word**
9. **Oral Reading**
10. **Reading Comprehension**
11. **Listening Comprehension**
12. **Orienting to Print**
13. **Number. I.D**
14. **Quantifying**
15. **Missing Number**
16. **Word Problems**
17. **Addition Level 1**
18. **Addition Level 2**
19. **Addition Level 3**
20. **Subtract Level 1**
21. **Subtract Level 2**
22. **Subtract Level 3**

Any activity used by RoboTutor exercises at least one of these skills and at most 4 of these skills. This also means that some activities can be interdependent with others.

CTA Table

This contains 22 columns, each column corresponding to 1 of the 22 KC's mentioned above. Under each of these KC columns, is a list of RoboTutor activities that exercise this particular KC. If an activity exercises more than 1 skill, there will be more than 1 occurrence in the CTA Table. In other words, the number of KC's exercised by an activity. Some of these activities also have a suffix of the form “__it_n” or “.n” after their activity names, implying their nth occurrence. Activities with different suffixes of this form are not to be counted as unique activities: they are the same. **Figure 1** contains a screenshot of the CTA Table for the first 6 rows first 5 skills.

Content Areas and the Code Drop 2 Matrices Table

RoboTutor has 3 content areas, namely: Literacy, Numeracy and Stories. RoboTutor follows area rotation while providing practice to users and asks activities following a cyclic rotation of activities from Literacy, Numeracy, Literacy, Stories (L-N-L-S). Each of these three content areas have a matrix associated with them: a literacy matrix, a numeracy matrix, and a stories matrix and these matrices are contained within the Code Drop 2 Matrices Table.

Figure 1: Sample of the CTA Table

Writing Letters	Writing a Word	Writing a Sentence	Write a Number	Letters
write.ltr.lc:all.asc.A..Z.11	write.wrd:lc.nonwrd.len-4.lev1	write.sen.copy.ltr:story_1.noerror.1.1	write.arith:ADD-1D-H.1__it_1	akira:all.ltr.lc:A..A.all.10.rand.say.39
write.ltr.lc:vow.asc.A..Z.5	write.wrd:lc.nonwrd.len-6.lev3	write.sen.copy.ltr:story_1.noerror.1.2	write.arith:ADD-1D-H.1__it_2	akira:all.ltr.lc:A..Z.all.24.asc.say.14
write.ltr.lc:dic:all.asc.A..Z.12	write.wrd:lc.nonwrd.len-8.lev5	write.sen.copy.ltr:story_1.noerror.1.3	write.arith:ADD-1D-H.1__it_3	akira:all.ltr.lc:B..B.all.10.rand.say.40
write.ltr.lc:dic:vow.asc.A..Z.6	write.wrd:lc.wrd.cat-animals.2	write.sen.copy.ltr:story_1.noerror.1.4	write.arith:ADD-2D-H.3	akira:all.ltr.lc:CH..CH.all.10.rand.say.41
write.ltr.lc:trc:all.asc.A..Z.10	write.wrd:lc.wrd.cat-body.4	write.sen.copy.ltr:story_2.noerror.2.1	write.arith:ADD-3D-H.5	akira:all.ltr.lc:D..D.all.10.rand.say.42
write.ltr.lc:trc:vow.asc.A..Z.4	write.wrd:lc.wrd.cat-food.1	write.sen.copy.ltr:story_2.noerror.2.2	write.arith:SUB-1D-H.2	akira:all.ltr.lc:E..E.all.10.rand.say.43

Each matrix under Code Drop 2 Matrices contains a matrix of activities (of that content area) in increasing order of difficulty based on a heuristic. Thus, as we proceed from top to bottom, and left to right, we move along successively difficult activities in a content area. The placement mode of RoboTutor places a user at a particular position for each content area. When the user enters promotion mode, the user transitions along the matrix based on observed %correct attempts on an activity. The observed %correct attempts for 3 more or attempts on the current activity is compared to fixed performance thresholds of RoboTutor based on which the tutor takes promotion decisions: BACK, SAME, NEXT or SKIP to update the user's position to the previous activity, stay at same activity, promote to next activity or the one thereafter. RoboTutor uses Lenient thresholds for lenient activities (write and math) which have lower thresholds for promotion compared to non-lenient activities. **Figure 2** contains a screenshot of the first 3 columns of **Code Drop 2 Matrices** for content area literacy.

story.parrot:ltr:A.rand.16	story.hear:UC_Vowel_Song_1	bpop.ltr:uc:A..A.vow.rand.all.stat.show.1
story.parrot:ltr:A.rand.40	bpop.ltr:lc:A..A.vow.rand.all.stat.show.11	story.hear:LC_Vowel_Song_2
story.hear:letters_alphabet_song	story.parrot:ltr:A.rand.16	bpop.ltr:uc:A..A.all.rand.all.stat.show.27
story.hear:letters_alphabet_song	story.parrot:ltr:A.rand.40	bpop.ltr:lc:A..A.all.rand.all.stat.show.78
story.hear:LC_Vowel_Song_1	bpop.wrd:ba..ba.syl.rand.all.stat.show.1	akira.syl:lc:say.ba..ba.noShow..1
story.hear:LC_Vowel_Song_1	bpop.wrd:cha..cha.syl.rand.all.stat.show.51	akira.syl:lc:say.cha..cha.noShow..18
story.hear:begin.wrd:ha.rand.1	bpop.wrd:begin.wrd:ha.show.1	story.echo:story_1
story.parrot:story_4	story.parrot:comm_pic:cat-food.rand.1	bpop.wrd:wrd:food:lc.rand.stat.show.1
story.parrot:story_7	spelling:datasource_2	story.parrot:HF.wrd:len-4.rand.1
story.parrot:story_13	story.parrot:HF.wrd:len-5.rand.2	story.echo:story_13
story.parrot:story_19	spelling:datasource_3	story.parrot:HF.wrd:len-6.rand.3
story.parrot:nonwrd:len-4.rand.1	story.parrot:story_25	bpop.wrd:nonwrd:4:lc.rand.stat.show.1
story.parrot:HF.wrd:len-7.rand.4	story.parrot:story_31	story.parrot:story_32
story.parrot:HF.wrd:len-8.rand.5	story.echo:story_37	story.parrot:story_38
story.parrot:nonwrd:len-8.rand.5	story.echo:story_43	story.parrot:story_44
story.parrot:HF.wrd:len-9.rand.6	story.echo:story_49	story.parrot:story_50
story.parrot:HF.wrd:len-10.rand.7	write.wrd:lc:wrd:len-10.13	bpop.wrd:wrd:10:lc.rand.stat.show.13
story.read:story_1	story.hear:nafasi	write.sen.copy.ltr:story_1.noerror.1.1

Figure 2: Code Drop 2 Matrices (Literacy Matrix)

Transactions table

The transactions table contains data logged at the level of attempts: each row of the table corresponds to an user's attempt on an item under some activity. The transactions table we used had around 67 columns: the most important ones in our context being student response, activity type, activity name, content area, student ID, skills exercised by the activity, etc. We use the transactions table of village 130 which has 42,010 attempts by 8 students. **Figure 3** (next page) contains a screenshot of the first few rows of the transactions table. Many columns have been omitted in this screenshot since there is a lack of space to display all of the 67 columns. Instead, the screenshot shows only the most important columns for our analyses.

Fitting the HotDINA Student Model

In this section, we discuss how we use the various types of RoboTutor data mentioned earlier to fit our student model: HotDINA. This fitted student model will then be instrumental in building the Simulated Student (Student Simulator). The code repository for this section is available at [Github link omitted for blind review] with documentation. However, running and reproducing this example requires full access to the RoboTutor data and is not uploaded on Github owing to storage constraints. The following are the steps we employ to fit HotDINA:

1. The raw transactions table contains many columns out of which only some are useful in the context of fitting the student model. Thus, the first step involves the extracting only the required number of columns for a given table. The file named **get_data_for_village_n.py** accomplishes this task. This script extracts the needed information given a village number and the number of attempts to be extracted. For example, running `python get_data_for_village_n.py -v 130 -o all` will extract all rows of village 130's transactions table. Running `python get_data_for_village_n.py -v 130 -o 500` instead, will extract the first 500 attempts (or rows) of the village 130's transactions table. Note that prior to running this script, all the transaction tables have to be located at `RoboTutor-Analysis/Data` (as mentioned in the README of the repository).
2. Once these scripts are run, the extracted details are stored as a pickle file as `hotDINA/pickles/data/data130_all.pickle` and `hotDINA/pickle/data/data130_500.pickle` respectively. **Figure 4** contains a screenshot of the extracted transactions data.
3. Since HotDINA requires a significant amount of computational resources for fitting with MCMC, we fit the model on a server (PSC Bridges in our case). Thus, we transfer the extracted data pickle files to the server by running `scp_data_to_bridges.py` to SCP (Secure Copy Protocol) all our pickle files before fitting. The Github documentation covers details on setup and running these SCP scripts.
4. We now have the extracted data in the server and can proceed to fit our student model. Our student model has 118 parameters in total: 8 parameters for student

Figure 3: Sample screenshot of the Transactions Table

Anon Student Id	Level (Tutor)	Problem Name	Outcome	CF (Matrix)	KC(Subtest)_1
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_D-J-Y	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_J-E-Y	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_K-O-J	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_P-E-J	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_M-N-J	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_J-W-P	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_A-E-J	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_CH-B-J	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_U-J-D	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_R-J-N	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_A-E-J	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_CH-B-J	INCORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_CH-B-J	INCORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_U-J-D	CORRECT	literacy	Syllable Sound
5A27001753	bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46	BPOP_J_J-E-Y	CORRECT	literacy	Syllable Sound

proficiency and 22 parameters for each of skill difficulty, skill discrimination, learn, guess and slip parameters. A more fulfilling (and more accurate)

version is to implement the full version of HotDINA which contains 8 student proficiency parameters and 1710 parameters for each of discrimination, difficulty, learn, guess and slip. Thus, the larger version contains $(8 + 1710 * 5)$ parameters. Due to computational constraints, we implement the smaller model which takes into account parameters by skill (22) rather than by activities (1710).

5. The `fit_hotDINA_skill.py` is a script written using PyStan to fit the HotDINA student model. This script uses 4 chains for MCMC sampling with a warm-up of 5000 and 15000 iterations. Fitting the model with these specifications took

- 241752 seconds. Since running programs on a supercomputer allocation like PSC Bridges requires the user to submit programs as jobs, there is a batch script named ‘run_hotDINA_skill.job’ which submits the fitting job to the server.
- Once the job finishes running, we get an out file named as ‘slurm-JOBID.out’ on the server. This is exactly the file containing all the fitted parameters for the HotDINA model. One can view the fitted model for this experiment at [Github link omitted for blind review]
 - This concludes fitting the student model for HotDINA and through this process, we obtain the fitted model parameters.

Figure 4: Extracted transactions data for first 3 rows of transactions table

```
items : [302, 302, 302]
users : [0, 0, 0]
y : [1, 1, 1]
skill_group_names : [['Letters', 'Syllable Sound'], ['Letters', 'Syllable Sound'], ['Letters', 'Syllable Sound']]
skill_group_nums : [[4, 5], [4, 5], [4, 5]]
activities : ['bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46', 'bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46', 'bpop.ltr.uc_J..J.all.rand.all.stat.noShow.46']
```

The **items** list in **figure 4** contains a list of **activity_number**-s for a given activity. The **activity_number** is defined in the following way: Suppose we have a list of all unique activities from the CTA (remove suffixes in the activity name, if any), the **activity_number** is defined as the index of the an activity into this list of unique activities.

The **users** list in **figure 4** contains a list of **student_num**-s. The first “Anon Student Id” encountered in the transactions table is user 0, the second “Anon Student Id” encountered is user 1 and so on.

The **y** list in **figure 4** is a list of student responses (1 for CORRECT and 0 for INCORRECT) after extraction from the transactions table.

The i^{th} element of **skill_group_names** is a list of skills names exercised by the i^{th} row of the transactions table.

The i^{th} element of **skill_group_nums** is a list of indices of skills exercised by the i^{th} row of the transactions table. The skill indices are 0-indexed with respect to the order of skills mentioned in page 1 (which is also the order of skills found in the CTA as from **Figure 1**).

The **activities** list in **figure 4** is a list of “Level (Tutor)” from the transactions table.

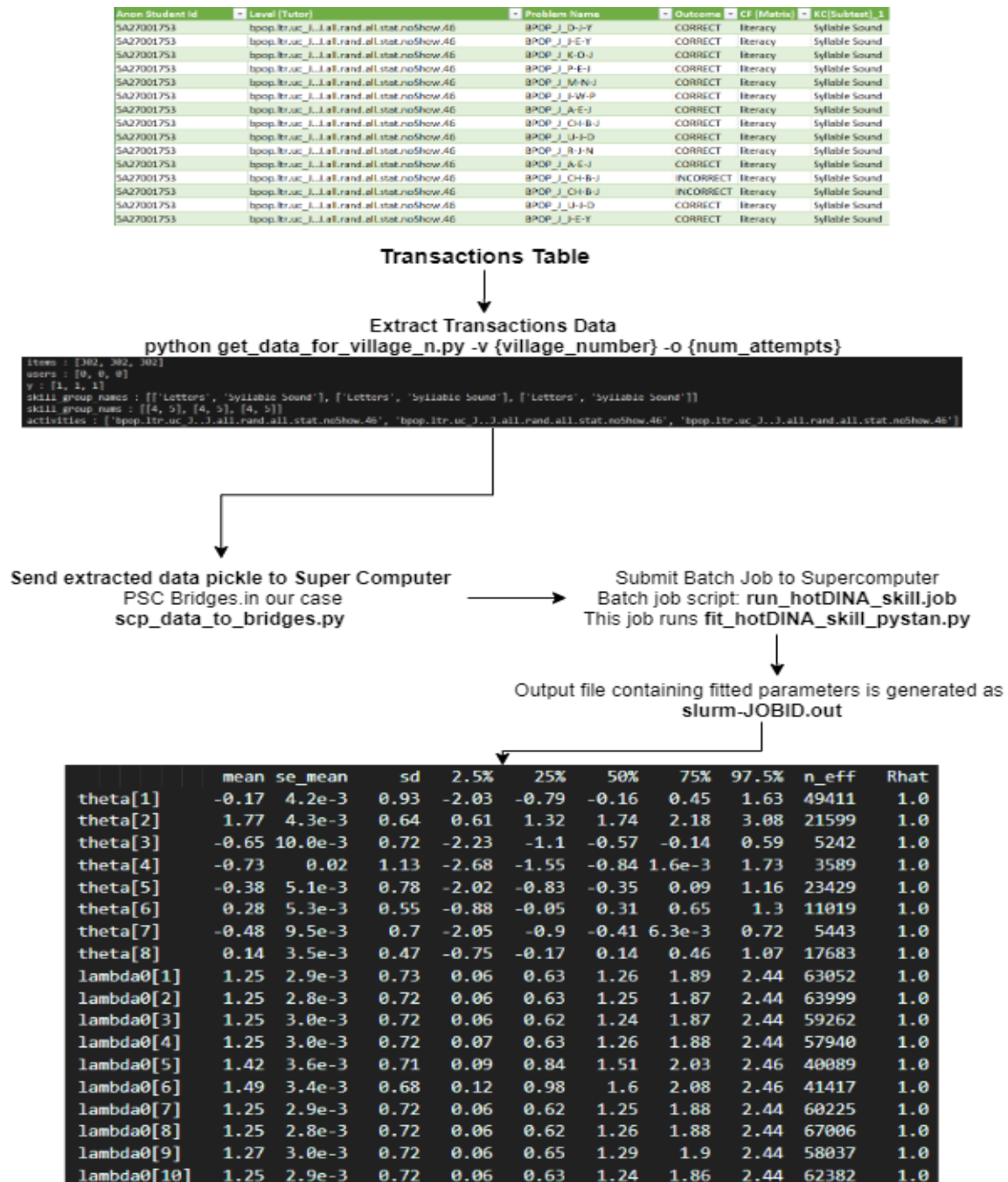
Figure 5 contains a screenshot of the fitted model parameters for our hotDINA student model. In the figure, we have 8 estimates for theta, implying one student proficiency parameter estimate for each of the 8 students from the transactions table of village 130. The **lambda0** parameter refers to skill difficulty and thus, we have 22 parameter estimates for **lambda0**. The figure has been cropped and contains only the converged skill difficulty parameter estimates for the first 10 skills. The full version can be seen in the HotDINA repository.

Figure 5: Fitted student model parameters for HotDINA

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
theta[1]	-0.17	4.2e-3	0.93	-2.03	-0.79	-0.16	0.45	1.63	49411	1.0
theta[2]	1.77	4.3e-3	0.64	0.61	1.32	1.74	2.18	3.08	21599	1.0
theta[3]	-0.65	10.0e-3	0.72	-2.23	-1.1	-0.57	-0.14	0.59	5242	1.0
theta[4]	-0.73	0.02	1.13	-2.68	-1.55	-0.84	1.6e-3	1.73	3589	1.0
theta[5]	-0.38	5.1e-3	0.78	-2.02	-0.83	-0.35	0.09	1.16	23429	1.0
theta[6]	0.28	5.3e-3	0.55	-0.88	-0.05	0.31	0.65	1.3	11019	1.0
theta[7]	-0.48	9.5e-3	0.7	-2.05	-0.9	-0.41	6.3e-3	0.72	5443	1.0
theta[8]	0.14	3.5e-3	0.47	-0.75	-0.17	0.14	0.46	1.07	17683	1.0
lambda0[1]	1.25	2.9e-3	0.73	0.06	0.63	1.26	1.89	2.44	63052	1.0
lambda0[2]	1.25	2.8e-3	0.72	0.06	0.63	1.25	1.87	2.44	63999	1.0
lambda0[3]	1.25	3.0e-3	0.72	0.06	0.62	1.24	1.87	2.44	59262	1.0
lambda0[4]	1.25	3.0e-3	0.72	0.07	0.63	1.26	1.88	2.44	57940	1.0
lambda0[5]	1.42	3.6e-3	0.71	0.09	0.84	1.51	2.03	2.46	40089	1.0
lambda0[6]	1.49	3.4e-3	0.68	0.12	0.98	1.6	2.08	2.46	41417	1.0
lambda0[7]	1.25	2.9e-3	0.72	0.06	0.62	1.25	1.88	2.44	60225	1.0
lambda0[8]	1.25	2.8e-3	0.72	0.06	0.62	1.26	1.88	2.44	67006	1.0
lambda0[9]	1.27	3.0e-3	0.72	0.06	0.65	1.29	1.9	2.44	58037	1.0
lambda0[10]	1.25	2.9e-3	0.72	0.06	0.63	1.24	1.86	2.44	62382	1.0

The Rhat values have a value of 1.0 indicating that the parameters have converged. If Rhat is not equal to 1.0, one might have to run the fitting process for a higher number of iterations, till convergence. **Figure 6** below summarizes in the form of a flow chart, the process for fitting the student model for the purpose of simulating students.

Figure 6: Fitting the Student Model



Part 1: Simulating students

Once we fit the parameters to the student model, simulating any of the 8 students from village 130 is a straightforward task. Let us consider the case of simulating the first student (Student 0) observed in the transactions table.

We can infer $\Pr(\text{Student 0 Knew skill } k)$ or $\Pr(\alpha_{0k}^{(0)})$ (in HotDINA paper) from the IRT parameters for HotDINA: skill discrimination (**lambda1[k]** in our code and b_k in the HotDINA paper), student proficiency (**theta[1]** or θ_n in the HotDINA paper), and skill difficulty (**lambda0[k]** in our code and a_k in HotDINA paper). We also have values for learn, guess and slip parameters per skill.

While simulating the responses of a student, we proceed as one normally would in a Knowledge Tracing setting. That is, for simulating a student response, we would calculate $\Pr(\text{Student 0 gets skill } k \text{ Correct})$ and do a coin flip based on this probability. The result of this coin flip is taken as CORRECT or INCORRECT response of the simulated student. The HotDINA paper covers the estimation of $\Pr(\text{Student 0 gets skill } k \text{ Correct})$ in more detail.

The parameters estimated so far by fitting, are also sufficient to do Bayesian knowledge tracing updates on HotDINA. Thus, we have covered the initializing a student model, performing Bayesian Knowledge Tracing (BKT) updates, and simulating a response for any student in the transactions table for village 130. The code for these (initializing a student model, performing BKT updates, and simulating response) can be found here: [Github link omitted for blind review]. The Student simulator itself, inherits the HotDINA student model and is present at: [Github link omitted for blind review]

Part 2: Training a Policy

We first instantiate the environment that the RL Agent will be interacting with. The environment takes as input, a Student Simulator which will interact with any activity posed to the environment, by the RL agent. For this section, most of the referred code will be in this repo: [Github link omitted for blind review]

We have seen earlier that the state space for any of the 4 types of RL agents involve 2 kinds of states: A knowledge state and a tutor state. The knowledge state is handled by the student model (**hotDINA_skill.py**) and the Student Simulator (**student_simulator.py**). Apart from the Student Simulator, we also have a Tutor Simulator (**tutor_simulator.py**) which takes different tutor decisions depending upon the agent type it is interacting with. The tutor simulator also keeps track of the tutor states. It can take various decisions for any given RL agent type. Here is an example of the Tutor Simulator's decisions for each RL agent type:

1. Promote or demote based on thresholds (type 1), update student position in content areas, keeping track of area rotation, and posing the next activity to the student simulator.
2. Promote or demote directly based on RL agent action (independent of thresholds), update student position in content areas, keeping track of area rotation, and posing the next activity to the student simulator.
3. Keeping track of area rotation and posing the next activity to the student simulator.
4. Posing the next activity to the student simulator.

Once an action is passed from the RL agent to the Student Simulator, it can respond to a given activity with either a 1 or a 0 (CORRECT or INCORRECT) based on coin flip of $\Pr(\text{Correct})$ which is being done in **hotDINA_skill.py**. After the student response is simulated, we use the student model to perform a Bayesian update for this response to the given activity. We now have a Prior(Know) and Posterior(Know). The RL agents uses the average of these differences as its reward function. At this point, 2 different events occur:

1. The tutor simulator uses Posterior(Know), student response, previous content area and previous matrix position to update the tutor state.
2. The Knowledge state is updated to the Posterior in the student model.

These updated values of knowledge state and tutor state serve as the next state for the RL agent. In this fashion, the RL agent and the environment (Student Simulator + Tutor Simulator) interact with each other.

We learn the policy using a state-of-the-art RL algorithm: Proximal Policy Optimization (a well-known policy-gradient method). To learn the ideal state-action mapping, we use a neural network with a depth of 2-4 layers depending upon the RL agent type that is being used. One can have a detailed look at the Neural Network architecture by referring to **RL_agents/actor_critic.py** in the mentioned repository.

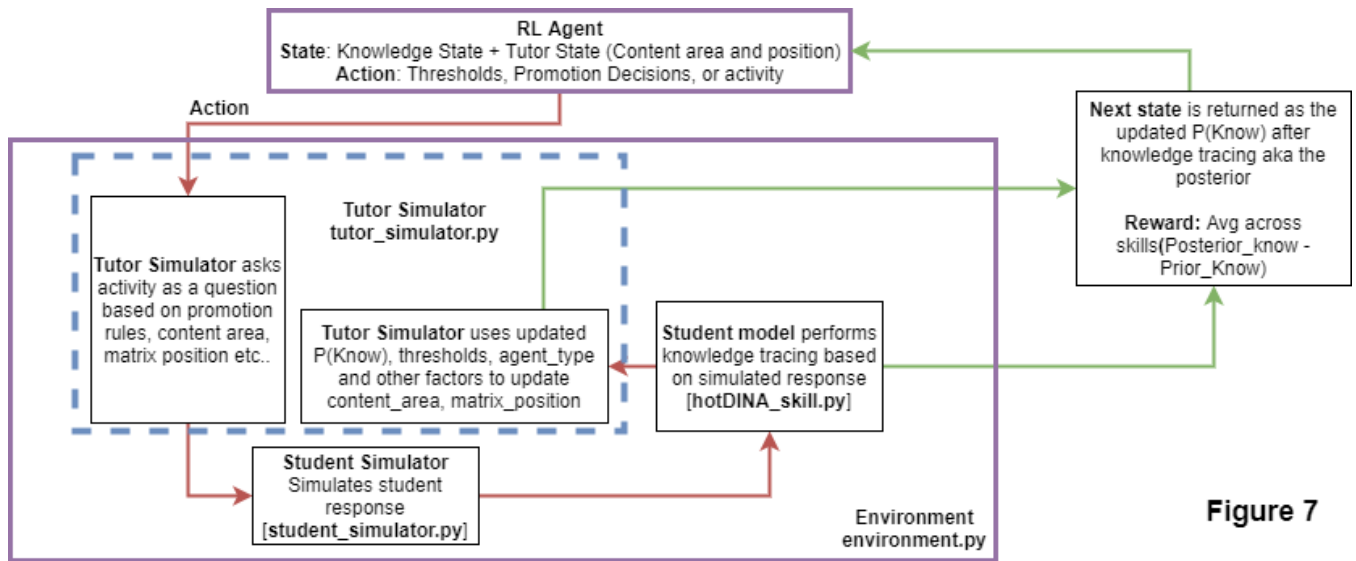


Figure 7

The implementational details of PPO, hyperparameters, as well as learning using PPO is in `learn_ppo_policy.py` and `RL_agents/ppo_helper.py` under the “RoboTutor-Analysis” repository.

Figure 7 illustrates the RL setup and the interaction between the RL Agent and the Environment (Student Simulator + Tutor Simulator). Due to computational expenses and in the interest of time, we trained the policy with each episode lasting 100 timesteps (attempts) and terminate episodes after they touch this number.

Thus, we cover the 2nd part of the STEP framework: Training a Policy.

Part 3: Evaluating Policies

In this section, we discuss the details of how we evaluate our policy. We evaluate our policies on not more than 100 attempts per student since the policy was not trained to work on larger attempts due to computational constraints. However, it is important to note that this policy is not overfitting to 100 attempts. A policy trained on 100 attempts would not be able to perform optimally on more than 100 attempts but can still perform well below 100 attempts. Thus, this is not a case of overfitting.

Tracing from Historical Data

We first evaluate the original policy employed by RoboTutor. We perform knowledge tracing on historical attempts to assess learning gains due to the historical policy. Code Repo: **RoboTutor-Analysis**. Code filename: `evaluate_policy.py`

Local Policy Evaluation

For every attempt ‘i’ on historical data, we take the prior(Know) due to the previous (i-1) attempts. We use this as a knowledge state and play our learned policy to look at the local impact if the RL policy had been played instead of RoboTutor’s current policy. For every first 100 attempts of a student in the transactions table, we obtain the local policy reward after each attempt ‘i’ as **Posterior(Know after applying RL policy for 1 attempt after i-1 historical attempts) – Prior(Know based on i-1 historical attempts)**. We take the average of local policy rewards across total number of attempts (8*100) and measure the percentage improvement compared to Posterior(After following historical trajectory). We term this as the local policy impact.

Global Policy Evaluation

For every first 100 attempts of a student in the transactions table of village 130, we run our learned policy over 100 attempts (or timesteps). After each attempt, we compare the Posterior(Know) obtained after following the RL trajectory and term this as global policy reward. We take the average global policy rewards across total number of attempts (8 * 100) and measure the percentage improvement compared to Posterior(After following historical trajectory). We term this as the global policy impact.

