Intelligent Tutor Systems

Introduction

In this assignment, you will be building an intelligent tutoring system that mathematically models a student's competence and then presents practice problems of the right difficulty to challenge the student appropriately. Such a system, if effective, can help increase access to education around the world, especially in areas where there is a shortage of human teachers. In addition, it allows each student to proceed at a pace that they feel comfortable with. A smart student could quickly progress to harder and more challenging problems while a beginner could take their time becoming comfortable first. Our goals with the intelligent tutor system will primarily be to:

1. **Quickly diagnose** a student's competence in a topic
2. **Continually track** the student's competence, as it grows over time

Of course, this is easier said than done. It is notoriously hard to understand a human, even for another human. Only a trained teacher, after multiple interactions with a student, is able to accurately judge how good a student is. The task before us is to model the student's brain mathematically, and that is even harder because of the variety of factors that affect competence that need to be captured to form an accurate picture. For example, we know that self-esteem affects a student's ability to perform in a topic, but how do we account for that in code? Naturally, someone with in-depth experience in psychology, neuroscience and machine learning might be better placed in tackling this problem. Nonetheless, we can still make significant strides to building a functional intelligent tutor system that does a decent job.

We will explore 2 main algorithms: the random approach and the reinforcement learning approach. Below you will find detailed explanations of the approaches, and you will be required to implement the approaches in Python within the given boilerplate code.

The goal of this assignment is to understand and implement an interesting algorithm. The end-product will be the core backend logic for such an intelligent tutor system, which you might choose to combine with a frontend and real practice problems in the future (if you feel like it) and then launch for real students or your friends to use! Even if you don't pursue that, you will have gotten your hands dirty with implementing math in code, which is a useful skill.

Random Approach

We randomly choose problems to show the student. This is the most obvious algorithm and we will show that it does not perform too well.

Reinforcement Learning Approach

We create 3 different student types and represent them as Markov Decision Processes, and then use Thompson Sampling to quickly discover which of these 3 types a new student fits.

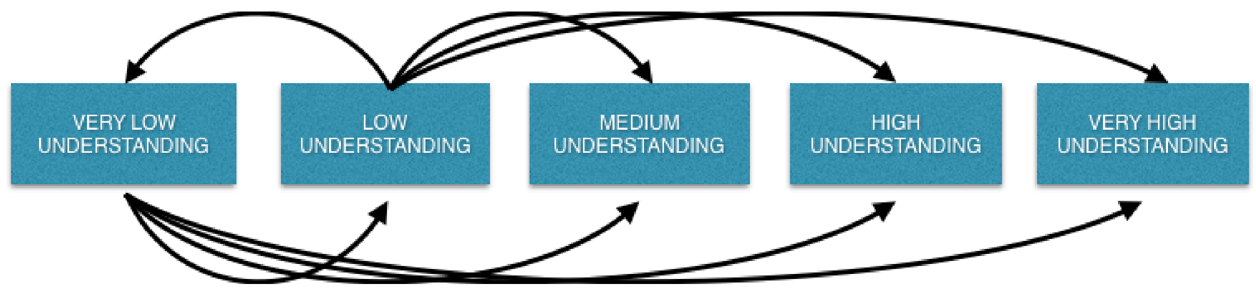
Brief Intro to Reinforcement Learning

Reinforcement learning (RL) is a branch of artificial intelligence concerned with training agents to explore a space of possibilities and automatically converge to the optimal choice to match a specified objective. For example, you could use RL to train a robotic arm to pick up a cup, by specifying what the objective is (i.e. cup is in its hand), and rewarding it for progress and penalizing it for mistakes made as it explores via trial-and-error. The agent learns to act in such a way that the reward is maximized, ultimately leading to successfully picking up the cup. The reward is generally a positive or negative number, where the former tells the agent it did a good action while the latter implies a bad action.

We will use techniques related to reinforcement learning in an interesting way in this assignment. In particular, we will train an agent (i.e. the intelligent tutor) to present the most appropriately challenging problem to a student. If it presents a good problem, we reward the agent, telling it to continue pursuing those types of problems. Else, we penalize the agent, so it knows it’s doing something wrong. Of course, there are a lot of ambiguities here. What defines a “good” or “bad” problem? How do we characterize a student’s competence? What does it mean for a problem to be “appropriately” challenging? After presenting a “bad” problem, how does the agent correct itself to avoid the mistake again? These are all design questions you would face in any general reinforcement learning problem. For this assignment, we answer these questions very simply. As you’ll see, even the simple way can be hard to understand or implement correctly, so it’s a great starting point. After this assignment, you might want to explore more nuanced ways of answering these questions to come up with a better agent! Ultimately, the goal is to build the best agent possible, where “best” is defined as an agent that most accurately models a student.

Framing a student type as a Markov Decision Process

We use a Markov Decision Process (MDP) to represent the 5 different levels of understanding that a student can have in a topic (say, algebra). An MDP looks like this:



Each box represents an understanding state a student could be in. If the student understands algebra really well, we might place her in the ‘Very High Understanding’ state, and then present problems of corresponding ‘Very High Difficulty’. We don’t know which state a new student is in, so we need to account for all possible states in our model, and then figure out which one a student is in as quickly as possible. The arrows (from each state to every other state, not all are shown) imply our model can update its belief of which state the student is in. Say we initially thought the student had ‘High Understanding’, but observed a few incorrectly answered problems in a row. Perhaps, we should then revise our belief to ‘Medium Understanding’ since we overestimated the student’s ability. In code, we simply have a variable that holds one of the 5 states, and we just update it if our belief changes. In short, the MDP captures what the agent thinks about a student at some snapshot in time.

Next, we come up with few different known student types, and call them BeginnerStudent, OKStudent and AdvancedStudent. A BeginnerStudent is likely to find herself in the Very Low Understanding state, whereas the AdvancedStudent is more likely to be in the Very High Understanding state. This gives rise to 3 unique initial MDPs. When a new student starts using the system, the agent can compare her to one of these types and quickly figure out whether the student is of Beginner, OK or Advanced type. You will be implementing these three student types in the assignment.

Choosing the best student type using Thompson Sampling

We use Thompson Sampling to pick one of these three student types (MDPs) based on the student’s performance on the first 5 problems. Given a student’s performance (unknown student type) on a problem, we compute the likelihood of obtaining that performance for a beginner, ok and advanced student, and update our beliefs over these student types. Performance on a problem can be either 0, 0.5 or 1. We perform these updates over 5 consecutive problems. The pseudocode for the algorithm is as follows:

Algorithm 1 Thompson Sampling over first 5 questions

```
1: $p_{advanced} \leftarrow \frac{1}{3}$
2: $p_{ok} \leftarrow \frac{1}{3}$
3: $p_{beginner} \leftarrow \frac{1}{3}$
4: $state \leftarrow \textit{VERY LOW UNDERSTANDING STATE}$
5: for $qn = 1$ to $5$ do
6:     $simulator \leftarrow \textbf{sample}_p\{advanced, ok, beginner\}$
7:     $action \leftarrow greedy\_action(simulator)$
8:     $reward \leftarrow do\_problem(action)$
9:     $Z \leftarrow \sum_{j \in \{advanced, ok, beginner\}} P(rew|j, action).p_j$
10:    $p_{advanced} \leftarrow \frac{P(rew|simulator,action)}{Z} \cdot p_{advanced}$
11:    $p_{ok} \leftarrow \frac{P(rew|simulator,action)}{Z} \cdot p_{ok}$
12:    $p_{beginner} \leftarrow \frac{P(rew|simulator,action)}{Z} \cdot p_{beginner}$
13:    $state \leftarrow action$
14: end for
```

Implementation

You will be implementing functions to define key parts of our intelligent tutor. In the code, each function has detailed comments explaining the intuition and how to implement them.

Main.py
Contains code to instantiate simulations for the two approaches, random and reinforcement learning. You will be creating different types of students and simulation configurations, and executing multiple simulations. We have provided boilerplate to print the results, and what to expect (if you do everything correctly). It is recommended to finish implementing all other files before coming to this file.

Run.py
Contains pre-written classes for simulating a Run, which is where we create a few digital students, show them some problems, and record the performances. This is the boilerplate to really verify our algorithms. The comments include expected results, which you should be able to obtain successfully. You will be implementing the core logic for choosing the problems randomly here, for the first approach. You will also implement the code to implement Thompson Sampling for the second approach.

Utils.py
Implement a simple function to greedily choose a problem to show the student, based on the student's current state (in the MDP).

Mdp.py
Contains the code for the internals of the MDP, including the definition of the environment and agent. Nothing to implement here!

Student.py
Contains the code for different types of students, AdvancedStudentConfig, OKStudentConfig, BeginnerStudentConfig and OKToAdvStudentConfig. You will be implementing how these students perform on different difficulty level problems. Once defined, we can then compare a new student to these known models. They give us a good starting point for what a student could be like.

Qlearning.py
Contains the code for the algorithm that we use to learn the underlying MDP for each student type. The algorithm is called Q-learning, and it has been implemented for you. You will be implementing functions that pre-compute the probabilities of obtaining a reward for a given difficulty level problem, for each student type. This information is then used during Thompson Sampling to identify how likely a reward to have originated from a student type, so we can update our probabilities over the three student types.

Results

The comments included in main.py have information on the expected results you should get after running that code.