KBAI
Fall 2015

Final Examination

Q1. *As you know the Piazza forum for this class is fairly busy. (As of now, the residential section with less than 100 students has more than 900 postings; the online section with less than 190 students has more than 6000 postings). Many of the postings are open-ended, novel and creative, but many fall under the category of "frequently asked questions": routine questions that, perhaps with small variations, are repeated semester after semester. Design an AI agent that can automatically answer routine, repetitive, frequently asked questions on the Piazza forum of the KBAI class. Assume that the AI agent has access to all postings on the Piazza forums for previous sections of this class - some 40,000 in all. You may want to visit the Piazza forum to remind yourself of some of the routine questions and the answers to them. If such an AI agent was available for future sections of the class, it would be able to promptly answer many of the FAQ on Piazza.*

Answer:

The problem has been described in Q1. For a busy forum with thousands of new posts, it is hard to avoid people posting repeatedly / similar questions. A simple strategy that has been widely used is to create a "frequently asked questions" (FAQ) list, and force everyone to read it before posting new posts. This approach partly solves the problem, but it loads extra work to people who wants to post new posts. And when the FAQ list increases, this approach will be less and less effective since nobody would like to follow the rule to read a list with more than one hundred questions. Here, I present how to use **Frame** to design an AI agent that can automatically answer FAQ.

How to store FAQ?

A Frame is an artificial intelligence data structure that used to store or represent knowledge. It uses slots to store values. The information of one typical question asked during registration can be stored as a frame (Table 1). It contains frame name: "faq_01", " and AKO (a kind of) defines its category, that it is one kind of "FAQ", and many slots name as Question, Answer, Key word … with specific values filled. For some slots it also has default values, e.g. has authority: True (we assume most FAQs are answered by authorities as instructors or TAs), Expiration date: None (we assume most FAQs will be valid until the day to be updated).

Table 1:

Slot	**Value**	**Type**
faq_01	-	(This Frame)
AKO	FAQ	(parent frame)
Question	"When will time-tickets be issued for Phase-II?"	(instance value)
Answer	"Dates can be found on the academic calendar on the Registrar's website. http://www.registrar.gatech.edu."	(instance value)
Key word 1	"when"	(instance value)
Key word 2	"time-tickets"	(instance value)
Key word 3	"issued"	(instance value)
Key word 4	"Phase-II"	(instance value)
has authority	True	(default value)
Expiration date	None	(default value)

Frame "faq_01" (Table 1) stores the information of a specific registration question, then how to use the frame structure to store information of different questions? To do that, we need know more general information of the compositions of FAQ, which is the parent frame of "faq_01". The parent frame, named "FAQ", is shown in Table 2. In the frame, "FAQ" is the frame name, the slot value of Question, Answer, and Key words can be any String objects. The frame "FAQ" has two default values, has authority: True (default) or other Boolean value and Expiration date: None or any Date objects.

Table 2:

Slot	**Value**	**Type**
FAQ	-	(This Frame)
Question	String	(parent frame)
Answer	String	(parent frame)
Key word 1	String	(parent frame)
Key word 2	String	(parent frame)
Key word 3	String	(parent frame)
…	String	(parent frame)
has authority	True (default) IF-ADDED	(default value)
Expiration date	None (default) IF-ADDED	(default value)

Due to the similarity between Frame and Object-Oriented Programming, I can create a class called FAQ, which has five attributes: question (a string), answer (a string), keywords (a list of strings), has_authority (a Boolean value) and expiration_date (None or a date object).

```
class FAQ(object):
    def __init__(self, question, answer, keywords, has_authority, expiration_date):
        self.question = question
        self.answer = answer
        self.keywords = keywords
        self.has_authority = has_authority
        self.expiration_date = expiration_date
```

The information in frame “faq_01” can be stored as a FAQ object:

```
faq_01 = FAQ( “When will …”, “Dates can be …”, [“when”, “time-tickets”, …], True, None)
```

The FAQ list of a forum e.g. Piazza can be stored as a list of FAQ objects, *FAQ_List = [faq_01, faq_02, … , faq_1001]*.

How to automatically answer FAQ?

Ideally, when any user submits a new post, the agent will run a background search to identify whether a similar question is already in the FAQ list. If the answer is no that the agent cannot find any similar FAQ, the agent will publish user’s new post in forum. If the answer is yes, the agent will return the similar FAQ to user and wait the user’s response (accept or deny the provided FAQ with answer). If the user accepts the FAQ and its answer, then the agent will cancel the user’s original post. If the user denies the provided FAQ, then the user’s the post will be published in forum.

To do that, the agent first needs a help function *get_keyWords*, which takes a question (a string) and returns a list of key words (a list of string). For example, it takes the question shown in Table 1, and should return a list of key words like [“when”, “time-tickets”, “issued”, “Phase-II”]. The help function can be very simple in python:

```
def get_keyWords(question):
    keyWords = [ ]

    # filter is list of strings that do not contain much valuable information
    filter = ["be", "is", "was", "were", "a", "an", "will", "for", ……]

    #split the question string to a list of strings
    words = question. split()

    #search key words in words
    for wd in words:
        if wd not in filter:
            keyWords.append(wd)

    return keyWords
```

After extracting key words from user's new post, the agent will run a search among all FAQ object's key words list to identify any match. The agent will first create a python dictionary to store the match scores of each FAQ object like *{faq_01: **0**, faq_02: **0**, faq_03: **0**, ……}*. After the search, the dictionary will be updated as *{faq_01: **3**, faq_02: **1**, faq_03: **2**, ……}*. Finally, the agent will return the best match (or top 3 matches) to the user.

Let's assume that one user posts a new post: *"Hi guys, anyone knows when I can get my time-tickets for my registration Phase-II?"*

The agent detects the new post, and extracts key words from the new post.

```
question = "Hi guys, anyone knows when I can get my time-tickets for my registration Phase-II?"
keyWords = get_keyWords(question)
```

The agent creates a python dictionary to store search results:

```
dict = {}

# iterate all FAQ objects in FAQ_List, initialize match scores to 0
for faq in FAQ_List:
    dict[faq] = 0
```

The agent then searches through all FAQ object's key words lists to identify matches, and update the python dictionary:

```
#iterate key words in key words list of the new post
for kw in keyWords:

    #iterate all FAQ object's keywords to identify matches
    for faq in FAQ_List:
        if kw in faq.keywords:
            dict[faq] += 1
```

The agent returns the FAQ object with the highest match scores, and prints out the answer.

```
score = -1

# iterate all FAQ objects in the dict
for faq in dict.keys():
    if score == -1:
        score = dict[faq]
        faqMatch = faq

    elif dict[faq] > score:
        score = dict[faq]
        faqMatch = faq
#print FAQ question and answer
print faqMatch.question, faqMatch.answer
```

Q2. *Again as you know, each term the students in the KBAI class produce a few hundred AI agents for addressing problems similar to those on the Raven's test of intelligence. Let us call them Student-Agents. By now we have several hundred Student-Agents. Some of these agents are quite good: they likely would give fairly decent performance on the Raven's test. However, none of the Student-Agents is perfect. Further, just like we can think of different kinds of problems on the Raven's test, we can think of different types of Student-Agents: some types of Student-Agents may be better suited to some kinds of problems, but not necessarily others. You may want to look at your own agents, the agents you have reviewed, as well as the exemplary agents in this class to remind yourself of some of the different kinds of Student-Agents. Design a meta-AI agent that can combine the capabilities of the hundreds of different types of Student-Agents such that the performance of the meta-AI agent on the Raven's test is superior to that of any of the individual Student-Agents. This meta-agent may invoke the right Student-Agent (or Student-Agents) for each different kind of problem. If such a meta-AI agent was available, students in future sections of the class will be able to observe its decisions and behaviors, and learn from it.*

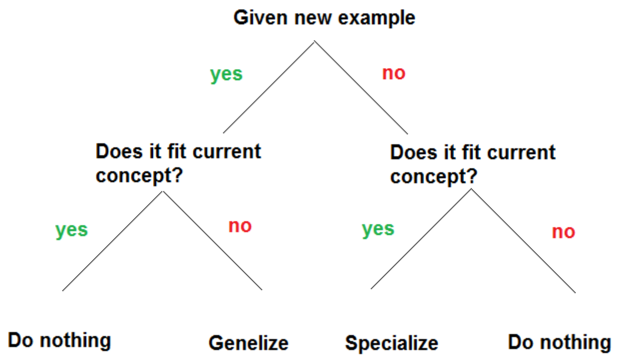
Answer:

If some student-Agents can solve Raven's test at 100% accuracy and the others solve Rave's test at 0% accuracy. The design of meta-AI agent will be very simple. However, the reality is no student-Agent is perfect. Some student-Agents are good at solving this set of Raven's tests, while others may be good at solving that set of Raven's tests. The meta-AI agent carries all advantages of its daughter agents, but at the same time inherits their defects as well. How to form a meta-AI agent with maximal advantages and minimal disadvantages, and how to dynamically adjust the composition of the meta-AI agent for different set of Raven's tests by removing bad-performing student-Agents and recruiting new good-performing student-Agents? Here, I present a strategy similar to **Incremental Concept Learning** to solve this problem.

What's incremental concept learning, and how to use that for meta-AI agent design?

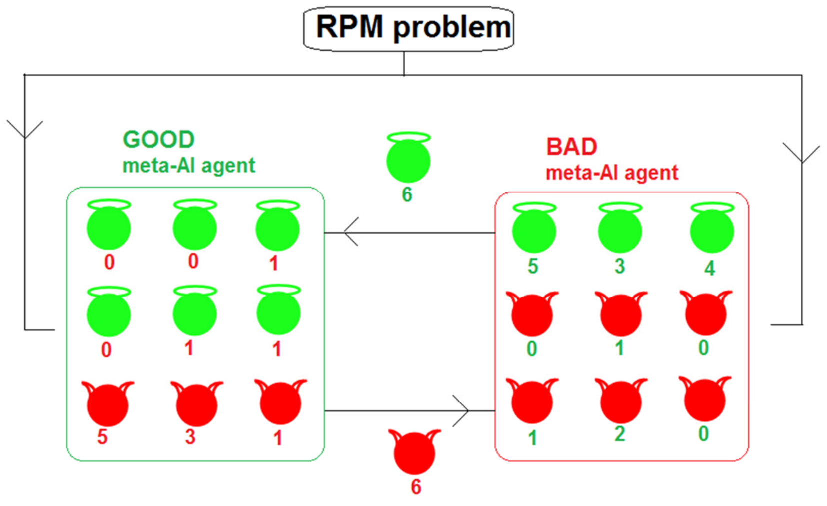
The concepts of objects are built on knowledge and experiences. One concept generates simultaneously when we see the first positive example. The new formed concept will be examined by more and more positive and negative examples (Figure 1). If we find positive examples that the concept does not include, we will generalize our concept to cover the new positive examples, while if we find negative examples that match the current concept, we will specialize our current concept to exclude the negative examples. The refinement procedure happens continuously; even on some well know concepts.

Figure 1: chart flow diagram of incremental concept learning.



The same strategy is used in designing meta-AI agent (Figure 2). Here, I have two concepts ("GOOD" meta-AI agent and "BAD" meta-AI agent) that need further refinements. These two concepts (meta-AI agents) are examined by giving new examples (different Raven's tests, RPM problems). Each RPM problem serves as a positive example for "GOOD" meta-AI agent. If "GOOD" meta-AI agent can solve the problem, do nothing. However, if the agent cannot solve the problem, we need start the refinement procedure. The same PRM problem will serve as a negative example for "BAD" meta-AI agent. If "BAD" meta-AI agent cannot solve the problem, do nothing. However, if the agent can solve the problem, we need start the refinement procedure.

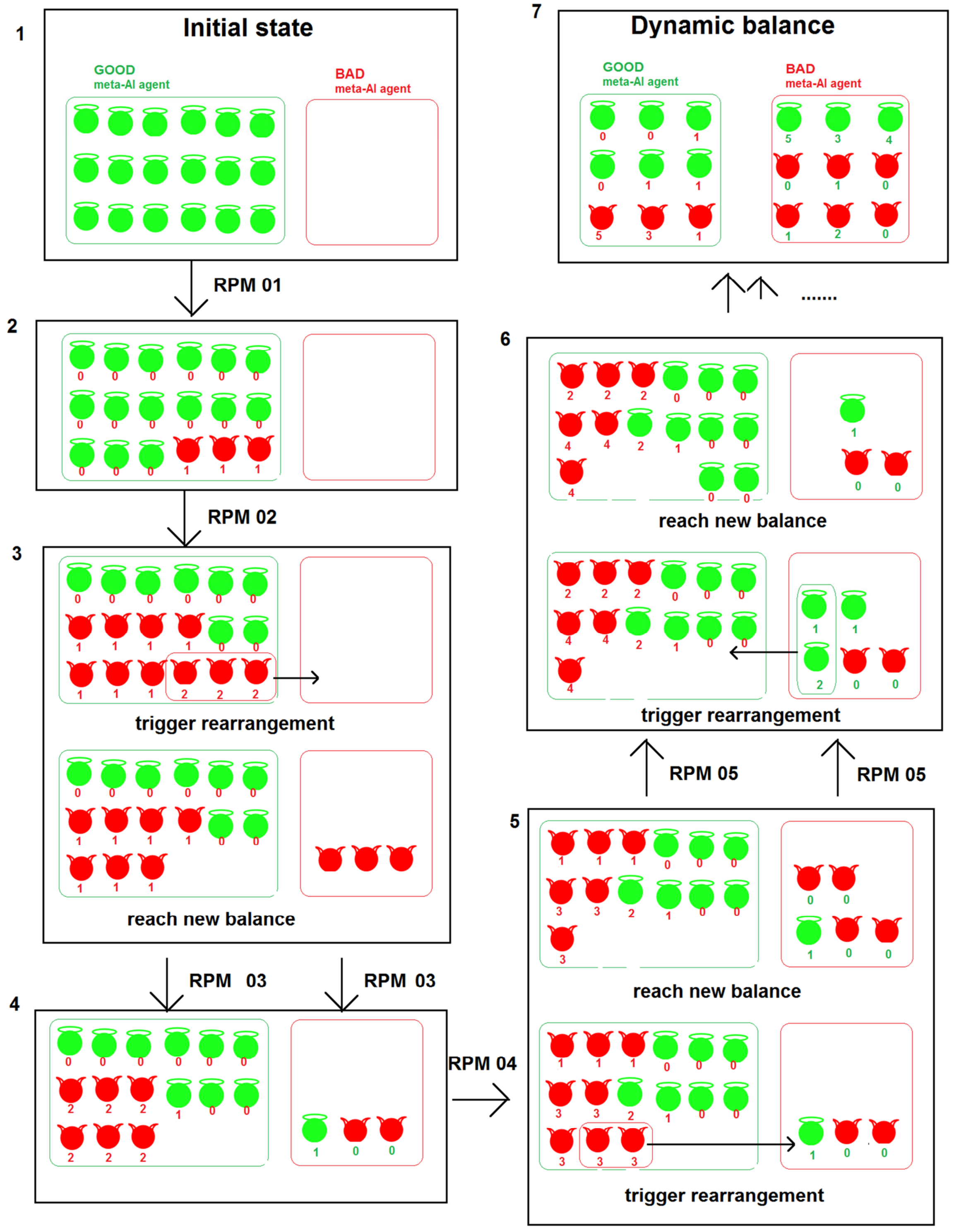
Figure 2: chart flow diagram of meta-AI agents' refinement



Since each meta-AI agents contain multiple student-Agents, the definition of “solve” will be a majority vote of all student-AI agents for any given RPM problem (e.g. more than 50% student-Agents give correct answers can be considered as the meta-AI agent solves the problem, the exact thresholds (percentiles) of “GOOD” or “BAD” meta-AI agents may be different). For the refinement procedure, every student-Agent in “GOOD” meta-AI agent will carry a number showing how many mistakes it has made since joining the “GOOD” campaign. The student-Agents having large mistake numbers will be purged with priority when the “GOOD” meta-AI agent fails a new test. Every student-Agent in “BAD” meta-AI agent also carries a number, but recording how many times it gives correct answers since joining the “BAD” campaign. The student-Agents having large correct numbers will be purged with priority when the “BAD” meta-AI agent passes a new test.

One detailed example is given in Figure 3. Let’s assume we have 18 student-Agents in total (the real number is a few hundred as described in Q2). At beginning, they are all recruited into the “GOOD” campaign (Figure 3.1). After examined by the first example (RPM 01), 3/18 student-Agents give wrong answers. These student-Agents gave wrong answers are labeled. And since the majority choose the correct answer (15 vs 3), no refinement is necessary (Figure 3.2). After examined by the second example (RPM 02), 10/18 student-Agents give wrong answers. Now the majority (10 vs 8) choose the wrong answers. The “GOOD” meta-AI agent faces a crisis, and triggers its purge process, which deports student-Agents with high mistake numbers into the “BAD” meta-AI agent campaign until the majority choose the correct answer (8 vs 7) (Figure 3.3). the “GOOD” and “BAD” meta-AI agents are both examined by RPM 03, and since the majority of the “GOOD” choose the right answer (9 vs 6) and the majority of the “BAD” choose the wrong answers (2 vs 1), no refiment is required (Figure 3.4). RPM 04 triggers another crisis for the “GOOD” meta-AI agent, and as result two student-Agents are deported into the “BAD” campaign (Figure 3.5). RPM 05 triggers a crisis for the “BAD” meta-AI agent since the majority choose the correct answer (3 vs 2), and as result two student-Agent are deported into the “GOOD” campaign (Figure 3.6). After a series of tests, the “GOOD” and “BAD” meta-AI agents reach a dynamic balance (Figure 3.7).

Figure 3: chart flow diagram of "GOOD" and "BAD" meta-AI agents' refinement process



The implementation

The “GOOD” and “BAD” meta-AI agents can be implemented as python dictionaries, using student-Agents as keys and mistake number (for “GOOD”) or correct number (for “BAD”) as values.

```
#Initialization:
GOOD_meta-AI = {student-Agent_01: 0, student-Agent_02: 0, student-Agent_03: 0, ……}
BAD_meta-AI = { }
```

After given a new RPM problem, both of meta-AI agents will run internal tests by their student-Agents and record the number of student-Agents that giving right answer versus student-Agents that giving wrong answers, and based on the majority vote to decide whether the refinement process is required. Both dictionaries are updated.

```
RPM = new RPM problem

#update GOOD_meta-AI
correctAgent = 0
wrongAgent = 0

for agent in GOOD_meta-AI.keys():
    if agent(RPM) gives correct answer:
        correctAgent += 1
    else:
        wrongAgent += 1
        #update student-Agent’s mistake number
        GOOD_meta-AI[agent] += 1

#if the majority choose wrong answers, start the purge procedure
while (wrongAgent > correctAgent):
    move student-Agent with highest value to BAD_meta-AI
    wrongAgent -= 1

#update BAD_meta-AI
correctAgent = 0
wrongAgent = 0

for agent in BAD_meta-AI.keys():
    If agent(RPM) gives correct answer:
        correctAgent += 1
        #update student-Agent’s correct number
        BAD_meta-AI[agent] += 1
    else:
        wrongAgent += 1
#if the majority choose the right answer, start the purge procedure
while (correctAgent > wrongAgent):
    move student-Agent with highest value to GOOD_meta-AI
    correctAgent -= 1
```

The advantages

There are other strategies to form a meta_AI. One simple example would be let all student-Agents run through one hundred RPM problems, and pick the top 20% student-Agents giving most right answers. What is the advantage of my approach? A big advantage of my approach is that my meta-AI agents can dynamically shape themselves based on different RPM problems. As discussed in Q2, different student-Agents have their advantages and disadvantages, e.g. some are good at solving 2X2 RPM while others are good at solving 3X3 RPM problems. If using the simple approach (choose top 20% performance), all student-Agents are given one hundred 2X2 RPM problems, and the formed meta-AI agent will only be good at 2X2 problems. Later on, when the meta-AI agent is given twenty 3X3 problems, the meta-AI agent will have poor performance and it is hard to adjust the compositions of the meta-AI agent since some student-Agents are so good at 2X2 problems, that even they have poor performance on 3X3 problems, they still have top 20% performance, thus stay in the meta-AI agent. My approach has no such problem.