Build a Simple Dialogue System whose Behavior is Driven by a Reinforcement Learned Policy

CSCI 544, Spring 2021: Assignment 2

Due Date: March 31st, 4pm.

Description:

For this assignment you are tasked with building a simple dialogue system that provides restaurant recommendations (text input, text output). We provide some details on each of the modules of this system below.

Dialogue Manager:

The dialogue manager (DM) is the heart of the dialogue system and it has a policy based on which it decides what the system should do next. You should have a simple frame-based DM trained using reinforcement learning (see below). The system should have three slots which are associated with the following information: type of food, price information, and location.

The possible values for each slot are:

<FOOD_TYPE>: empty | any | Italian | Japanese | Chinese | Mexican | Greek
<PRICE>: empty | any | cheap | medium-priced | expensive
<LOCATION>: empty | any | Marina Del Rey | Venice | Santa Monica | Korea Town | Playa Vista | Hollywood

Initially at the beginning of the dialogue all the slots have the "empty" value.

The DM should query for the food type (e.g., "what type of food do you want?"), the price (e.g., "how expensive a restaurant do you want?"), and the location (e.g., "where do you want the restaurant to be located?"). The order of queries doesn't matter (more details about this are provided below).

The DM should have an explicit confirmation policy that confirms every slot value provided by the user. For example, if the user requests "Italian" food then the system explicit confirmation request would be "did you say Italian?". Thus, for each slot there should also be a confirmation slot that must be filled in positively. So, for the slot FOOD_TYPE there should also be a slot FOOD_TYPE_CONF, for the slot PRICE there should also be a slot PRICE_CONF, and for the slot LOCATION there should also be a slot LOCATION_CONF. If for example FOOD_TYPE has a value and the user has positively confirmed this value then "FOOD_TYPE_CONF=yes". If FOOD_TYPE has a value and the user has not confirmed this value then "FOOD_TYPE_CONF=no". In the case of negative confirmation (i.e., when the user responds "no" to the system confirmation request) "FOOD_TYPE_CONF=no" and "FOOD_TYPE" should now become "empty", which means that the system will have to request information about the food type again (but not necessarily in the next system turn). Likewise for the rest of the slots. **Note that the system should only try to confirm slots that have been filled.**

So, the DM can choose among 6 possible system actions: REQUEST_FOOD_TYPE, REQUEST_PRICE, REQUEST_LOCATION, EXPLICIT_CONFIRM_FOOD_TYPE, EXPLICIT_CONFIRM_PRICE, EXPLICIT_CONFIRM_LOCATION.

The order of filling or confirming slots doesn't matter. For example, the system could first request the price, then request the food type, then confirm the food type, then confirm the price, then request the location, and then confirm the location. Or, it could first request the food type, then request the price, then request the location, then confirm the location, then confirm the food type, and then confirm the price. Or, it could first request the location, then confirm the location, then request the food type, then confirm the food type, then request the price, and then confirm the price. And so forth and so on. Any order is acceptable as long as a confirmation request for a slot comes after the slot has been filled.

The DM should be able to handle cases where the user provides irrelevant information or doesn't say anything. For example, if the system asks "what kind of food would you like?" and the user says nothing or says "no idea", then the system should consider this input as irrelevant and ask the user again. The system is not expected to handle cases where the user provides more information than what is requested. For example, if the system says "what price range?" and the user responds "cheap and in Playa Vista" the system should ignore the location (Playa Vista) because it asked for the price only. When the system asks the user to confirm the value of a slot (e.g., "did you say cheap") it should be able to process "yes", "no", and empty or irrelevant responses by the user. So, if the user says "yes" the slot will be confirmed, if the user says "no" the slot will not be confirmed and its value will become "empty", and if the user says nothing or something irrelevant then the user response will be ignored (the dialogue state won't change), which means that the system will have to ask again for confirmation.

After all the slots are filled and confirmed the DM should query a database that is provided in a tab separated value file consisting of five columns:
<RESTAURANT_NAME> \t <RESTAURANT_PHONE_NUMBER> \t <FOOD_TYPE> \t <PRICE> \t <LOCATION>

The DM should then select all restaurants that match the user's stated preferences and output the total number of restaurants that match, the names of the restaurants, and their phone numbers. Note that the value *any* indicates that all values for that category are acceptable to the user and thus no restaurants should be filtered from the results for that value.

So, the above description shows how the DM should behave and in order to build such a DM you'll use reinforcement learning to learn the DM policy.

Reinforcement Learning:

The reinforcement learning policy should learn to request information about each slot and then confirm this slot (querying the database is not part of the reinforcement learning problem). So, for the reinforcement learning problem the state space can be simplified. You only need the following state variables (you don't need the actual values of the slots):

<FOOD_TYPE_FILLED>: no | yes
<PRICE_FILLED>: no | yes
<LOCATION_FILLED>: no | yes

<FOOD_TYPE_CONF>: no | yes
<PRICE_CONF>: no | yes
<LOCATION_CONF>: no | yes

Initially at the beginning of the dialogue all these state variables have the “no” value (nothing is filled or confirmed).

There are 6 possible system actions: REQUEST_FOOD_TYPE, REQUEST_PRICE, REQUEST_LOCATION, EXPLICIT_CONFIRM_FOOD_TYPE, EXPLICIT_CONFIRM_PRICE, EXPLICIT_CONFIRM_LOCATION.

To train the policy you need a simulated user. The simulated user should exhibit the behavior we want from a real user. As we discussed above, we need the system to be able to fill and confirm the slots, and also handle cases where the user provides irrelevant information or says nothing. For this reason, the simulated user should be able to generate the following actions: PROVIDE_FOOD_TYPE, PROVIDE_PRICE, PROVIDE_LOCATION, YES_ANSWER, NO_ANSWER, IRRELEVANT. IRRELEVANT corresponds to an empty or irrelevant user response.

The simulated user should provide the information requested by the system or say “IRRELEVANT” (empty or irrelevant response) with a probability distribution (the exact probabilities are up to you). For example, if the system action is “REQUEST_FOOD_TYPE” then the simulated user response could be “PROVIDE_FOOD_TYPE” with probability 0.6 and “IRRELEVANT” with probability 0.4. In this case neither the probability of “PROVIDE_FOOD_TYPE” nor the probability of “IRRELEVANT” should be 0 because if you do that the resulting learned policy won’t be able to handle all required cases. Also, in this case you don’t need to assign probabilities for the user actions “PROVIDE_PRICE” or “PROVIDE_LOCATION” because we don’t expect the system to be able to handle cases where the user provides more information than what is requested. Likewise for the rest of the slots.

The simulated user should respond “YES_ANSWER”, “NO_ANSWER”, or “IRRELEVANT” (empty or irrelevant response) to the system’s confirmation request based on a probability distribution (the exact probabilities are up to you). For example, if the system action is “EXPLICIT_CONFIRM_FOOD_TYPE” then the simulated user response could be “YES_ANSWER” with probability 0.4, “NO_ANSWER” with probability 0.4, and “IRRELEVANT” with probability 0.2. In this case none of the probabilities for “YES_ANSWER”, “NO_ANSWER”, or “IRRELEVANT” should be 0 because if you do that the resulting learned policy won’t be able to handle all requested cases. Likewise for the rest of the slots. Again, when the simulated user responds to a system confirmation request you don’t need to assign probabilities to actions such as “YES_ANSWER, PROVIDE_PRICE” or “PROVIDE_PRICE”, etc. because after a confirmation request we expect the system only to be able to handle “YES_ANSWER”, “NO_ANSWER”, or “IRRELEVANT” responses (nothing else).

The reward function is as follows: 500 points when all slots are both filled and confirmed and -5 points for every system action. These are the only rewards/penalties you should use. **Do not change the reward function.**

For reinforcement learning you have to use the Q-learning algorithm with ε-greedy exploration. The exploration rate could be fixed or you could start with a fixed value and decrease it over time. So, you could select a fixed exploration rate of e.g. 20%, which means that the system should exploit 80% of the time and then pick randomly among all actions 20% of the time. Remember, exploitation means that the system selects at a state the action with the highest Q-value for that state. Or, you could start with a fixed exploration rate (e.g., 30%) and then decrease it gradually (e.g., by 0.3% every 10 episodes). How you do exploration is up to you. Note that how much exploration you'll need will also depend on how you implement the simulated user.

The value of γ should be 0.99. The learning rate α is equal to 1/(1+current_episode_number). **Do not change the value of γ and the formula for calculating α.** Each episode should time out after a number of system actions (this number is up to you, don't choose a very small or very large value, recommended numbers are between 20 and 30).

The number of episodes that the policy will need to converge will depend on your exact configuration but you should train it for at least 500 episodes (you may need to train even more e.g., for 1000 episodes). Also, it is possible that sometimes you'll get lucky and the policy will converge faster (e.g., after 400 episodes). You should do the training multiple times to make sure that the number of episodes you select for training is sufficient and that the resulting policy is always correct. Because of the probabilistic nature of the task there will be variations every time you train. You may also get different policies every time you train. **Remember, the order of handling the slots (e.g., whether price is handled before location or vice versa) does not matter. It also doesn't matter if the policy first tries to fill all the slots and then asks for confirmation, or if requests for filling slots and confirmations alternate. But the policy should learn first to fill a slot and then try to confirm it. It should never try to confirm a slot that is not filled.** The reason the order doesn't matter is because the rewards we chose do not pose such constraints.

Below we can see an example interaction between the policy and the simulated user during training. We can also see the dialogue state updates (more examples are given in the lecture slides explaining this assignment):

State: FOOD_TYPE_FILLED=no, PRICE_FILLED=no, LOCATION_FILLED=no, FOOD_TYPE_CONF=no, PRICE_CONF=no, LOCATION_CONF=no
Policy: EXPLICIT_CONFIRM_PRICE (this seems like a weird action at this point but it could happen as a result of exploration)
Simulated user: NO_ANSWER (IRRELEVANT, YES, ANSWER, and NO_ANSWER are the only possible simulated user actions here and NO_ANSWER was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=no, PRICE_FILLED=no, LOCATION_FILLED=no, FOOD_TYPE_CONF=no, PRICE_CONF=no, LOCATION_CONF=no
Policy: REQUEST_LOCATION
Simulated user: IRRELEVANT (IRRELEVANT and PROVIDE_LOCATION are the only possible simulated user actions here and IRRELEVANT was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=no, PRICE_FILLED=no, LOCATION_FILLED=no, FOOD_TYPE_CONF=no, PRICE_CONF=no, LOCATION_CONF=no

Policy: REQUEST_LOCATION
Simulated user: PROVIDE_LOCATION (IRRELEVANT and PROVIDE_LOCATION are the only possible simulated user actions here and PROVIDE_LOCATION was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=no, PRICE_FILLED=no, LOCATION_FILLED=yes, FOOD_TYPE_CONF=no, PRICE_CONF=no, LOCATION_CONF=no
Policy: REQUEST_PRICE
Simulated user: PROVIDE_PRICE (IRRELEVANT and PROVIDE_PRICE are the only possible simulated user actions here and PROVIDE_PRICE was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=no, PRICE_FILLED=yes, LOCATION_FILLED=yes, FOOD_TYPE_CONF=no, PRICE_CONF=no, LOCATION_CONF=no
Policy: EXPLICIT_CONFIRM_PRICE
Simulated user: YES_ANSWER (IRRELEVANT, YES, ANSWER, and NO_ANSWER are the only possible simulated user actions here and YES_ANSWER was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=no, PRICE_FILLED=yes, LOCATION_FILLED=yes, FOOD_TYPE_CONF=no, PRICE_CONF=yes, LOCATION_CONF=no
Policy: EXPLICIT_CONFIRM_LOCATION
Simulated user: NO_ANSWER (IRRELEVANT, YES, ANSWER, and NO_ANSWER are the only possible simulated user actions here and NO_ANSWER was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=no, PRICE_FILLED=yes, LOCATION_FILLED=no, FOOD_TYPE_CONF=no, PRICE_CONF=yes, LOCATION_CONF=no
Policy: REQUEST_FOOD_TYPE
Simulated user: PROVIDE_FOOD_TYPE (IRRELEVANT and PROVIDE_FOOD_TYPE are the only possible simulated user actions here and PROVIDE_FOOD_TYPE was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=yes, PRICE_FILLED=yes, LOCATION_FILLED=no, FOOD_TYPE_CONF=no, PRICE_CONF=yes, LOCATION_CONF=no
Policy: REQUEST_LOCATION
Simulated user: PROVIDE_LOCATION (IRRELEVANT and PROVIDE_LOCATION are the only possible simulated user actions here and PROVIDE_LOCATION was chosen based on the probability distribution you defined)
Reward: -5
State: FOOD_TYPE_FILLED=yes, PRICE_FILLED=yes, LOCATION_FILLED=yes, FOOD_TYPE_CONF=no, PRICE_CONF=yes, LOCATION_CONF=no
Policy: EXPLICIT_CONFIRM_LOCATION
Simulated user: YES_ANSWER (IRRELEVANT, YES, ANSWER, and NO_ANSWER are the only possible simulated user actions here and YES_ANSWER was chosen based on the probability distribution you defined)
Reward: -5

State: FOOD_TYPE_FILLED=yes, PRICE_FILLED=yes, LOCATION_FILLED=yes, FOOD_TYPE_CONF=no, PRICE_CONF=yes, LOCATION_CONF=yes
Policy: EXPLICIT_CONFIRM_FOOD_TYPE
Simulated user: YES_ANSWER (IRRELEVANT, YES, ANSWER, and NO_ANSWER are the only possible simulated user actions here and YES_ANSWER was chosen based on the probability distribution you defined)
Reward: -5 + 500
State: FOOD_TYPE_FILLED=yes, PRICE_FILLED=yes, LOCATION_FILLED=yes, FOOD_TYPE_CONF=yes, PRICE_CONF=yes, LOCATION_CONF=yes
Now the dialogue stops because we have reached the desired state where all slots are both filled and confirmed, hence the reward of 500.
Total reward at the end of this episode (dialogue): 450

After training, the learned dialogue policy should be saved in a file using the following format (comma-delimited) where each line corresponds to a state and the best action for that state: <FOOD_TYPE_FILLED>,<PRICE_FILLED>,<LOCATION_FILLED>,<FOOD_TYPE_CONF>,<PRICE_CONF>,<LOCATION_CONF>,<BEST_ACTION_FOR_THIS_STATE>.The file should contain all possible states in the following order (note that the file should also have a header):

```
FOOD_TYPE_FILLED, PRICE_FILLED, LOCATION_FILLED, FOOD_TYPE_CONF, PRICE_CONF, LOCATION_CONF, BEST_ACTION
0, 0, 0, 0, 0, 0, REQUEST_FOOD_TYPE
0, 0, 0, 0, 0, 1, REQUEST_PRICE
0, 0, 0, 0, 1, 0, REQUEST_FOOD_TYPE
0, 0, 0, 0, 1, 1, REQUEST_LOCATION
0, 0, 0, 1, 0, 0, EXPLICIT_CONFIRM_LOCATION
0, 0, 0, 1, 0, 1, REQUEST_PRICE
0, 0, 0, 1, 1, 0, EXPLICIT_CONFIRM_PRICE
0, 0, 0, 1, 1, 1, REQUEST_FOOD_TYPE
0, 0, 1, 0, 0, 0, REQUEST_PRICE
0, 0, 1, 0, 0, 1, REQUEST_FOOD_TYPE
0, 0, 1, 0, 1, 0, EXPLICIT_CONFIRM_PRICE
0, 0, 1, 0, 1, 1, REQUEST_PRICE
0, 0, 1, 1, 0, 0, REQUEST_FOOD_TYPE
0, 0, 1, 1, 0, 1, REQUEST_FOOD_TYPE
0, 0, 1, 1, 1, 0, REQUEST_FOOD_TYPE
0, 0, 1, 1, 1, 1, REQUEST_PRICE
etc.
```

Note that in the above file the best actions are just examples. Your policy could result in different best actions for those states. **The best action for a state is the action with the highest Q-value for that state. Thus, it is critical that you implement the Q-learning algorithm and calculate the Q-values for each state-action pair correctly. The Q-values dictate the policy.**

Note also that for the states which include slots that are confirmed but not filled it shouldn’t matter what the best action is. Because these states shouldn’t occur. The action could be any of the 6 actions or you could just use the “NULL” value. So, the states in lines 2-8 and 11-16 above (not counting the header) will never occur because in all these states we have cases where a slot is confirmed but not filled. For example, in line 16, the slots food type and price are confirmed but not filled. For the 1st state “REQUEST_FOOD_TYPE” is a possible best action, for the 9th state “REQUEST_PRICE” is a possible best action, and for the 10th state “REQUEST_FOOD_TYPE” is a possible best action. Alternatively you can save in the policy file only the states that occur. But you need to keep the header, the order of the state variables, and the order of the states. This will help the Professor and the TAs read your policy file more easily, if needed.

Your program should also save a comma-delimited file with information about the total reward per episode (dialogue) where each line corresponds to a training episode and has the current format:
<CURRENT_EPISODE_NUMBER>, <TOTAL_REWARD_AT_END_OF_THIS_EPISODE>
The information in this file will help you track whether the policy converges or not.

Important note: During training the policy you don’t need to worry about natural language understanding, natural language generation, or querying the database. All you need to care about is generating dialogues between the policy and the simulated user using dialogue acts such as the example shown above, and using the Q-learning algorithm to learn the Q-values for each state-action pair.

During testing, your program should read the dialogue policy from the saved file and use this policy to drive the system behavior. **During testing there is no further training of the policy, the policy is fixed.** Now you are going to need natural language understanding and generation, and will also have to query the database. For instructions on how to handle natural language understanding and generation see below. As we discussed above, the goal of the system is to fill and confirm the slots. After all the slots are both filled and confirmed the DM should query the database (see the DM section above).

Natural Language Understanding:

Natural language understanding should be performed using keyword matching. The expected key words have been listed for each slot above in the DM section. For confirmation queries the keywords are “yes” and “no”. If the user provides information that is not requested or says something irrelevant then the system should ignore it.

The system should be able to handle inputs with punctuation and lower or upper case letters. It should also be able to handle full sentences. For example, it should be able to handle all inputs below:

No
no, I don’t think so.
Playa Vista
playa vista
I want italian food
something Italian.
yes
medium-priced
medium priced food
etc.

The system is not expected to be able to handle alternative forms of “yes” (e.g., “yeah”) or “no” (e.g., “nope”). Also, the system is not expected to be able to handle negative statements such as “I don’t want Italian food”.

Natural Language Generation:

Natural Language Generation should be template-based and your confirmation utterances should contain the hypothesized user-provided values. Exact wording for each query isn’t specified but each query must be understandable enough to an English speaker that it can be easily associated with the expected dialogue act.

For example, all system prompts below are acceptable:

What kind of food would you like?
tell me what food type you want
did you say cheap?
You said cheap, is this correct
you want expensive food, right?
what price range are you interested in?
how expensive would you like the restaurant to be
etc.

Here are two complete example dialogues that your system should be able to handle (the exact order of system prompts will depend on the learned policy):

Example 1
System: What type of food do you want?
User: hello there
System: What type of food do you want? (note here that the system asks the same question, which is to be expected because after the irrelevant user response the state hasn’t changed, so the policy is going to select the same action, there is no training here, this is during testing and the policy is fixed)
User: I want Japanese food.
System: How expensive a restaurant do you want?
User: I want something medium-priced.
System: Ok you said you wanted a Japanese restaurant, right?
User: Yes.
System: Ok you said you wanted a medium-priced restaurant, right?
User: no
System: How expensive a restaurant do you want?
User: Something cheap
System: Ok you said you wanted a cheap restaurant, right?
User: yes
System: Ok where do you want the restaurant to be located?
User: marina del rey
System: Ok you said you wanted a restaurant in Marina Del Rey, right?
User: yes, that’s what I said
System: I found 1 restaurant matching your query. Tsujitan Artisan Noodles is a cheap Japanese restaurant in Marina Del Rey. The telephone number is 310-235-4352.

Example 2
System: How expensive a restaurant do you want?
User: Good question.
System: How expensive a restaurant do you want? (note here that the system asks the same question, which is to be expected because after the irrelevant user response the state hasn't changed, so the policy is going to select the same action, there is no training here, this is during testing and the policy is fixed)
User: Something expensive in Hollywood. (note that here Hollywood will be ignored because the system asked for the price only)
System: What type of food do you want?
User: Maybe something.
System: What type of food do you want? (note here that the system asks the same question, which is to be expected because after the irrelevant user response the state hasn't changed, so the policy is going to select the same action, there is no training here, this is during testing and the policy is fixed)
User: any
System: Ok you said you wanted any type of food restaurant, right?
User: Yes.
System: Ok you said you wanted an expensive restaurant, right?
User: what?
System: Ok you said you wanted an expensive restaurant, right? (note here that the system asks the same question, which is to be expected because after the irrelevant user response the state hasn't changed, so the policy is going to select the same action, there is no training here, this is during testing and the policy is fixed)
User: yes
System: Where would you like the restaurant to be located?
User: in hollywood
System: Ok you said you wanted a restaurant in Hollywood, right?
User: Yes.
System: I found 3 restaurants matching your query. Hiroshima is an expensive Japanese restaurant in Hollywood. The phone number is 310-356-7264. Mr. Changs is an expensive Chinese restaurant in Hollywood. The phone number is 310-535-7545. Canyon Road is an expensive Mexican restaurant in Hollywood. The phone number is 310-235-4636.

What to turn in and Grading

The command for training the policy should be "python system_train.py". This program should generate the files "policy.csv" and "rewards.csv". The number of training episodes should be included in your code so that when the Professor and the TAs train your system it will be trained for the number of episodes you have selected. The command for testing the system should be "python system_test.py". When you test the system the program should be able to read the "policy.csv" file and use it to drive the behavior of the dialogue system.

We'll train your system again but we also want you to upload the policy and rewards files. So that we don't overwrite your files you'll need to submit these files renamed to "policy-submitted.csv" and "rewards-submitted.csv" respectively.

You need to submit the following on Vocareum:

- **All your code**: system_train.py, system_test.py, and any additional code you created for the assignment. This is required and the academic honesty policy (see rules below) applies to all your code.
- **The files generated during training:** “policy-submitted.csv” and “rewards-submitted.csv”.
- **The database file**: “restaurantDatabase.txt”.

Note that the restaurant database is in Unix format.

We will grade your dialogue system based on some test cases we have created. These test cases are similar to the examples above and the examples in the lecture slides. If we are unable to execute your code successfully, you will not receive any credits. You will get partial credit based on the percentage of test cases that your program gets right.

Late Policy

- On time (no penalty): before March 31st, 4pm.
- One day late (10 point penalty): before April 1st, 4pm.
- Two days late (20 point penalty): before April 2nd, 4pm.
- Three days late (30 point penalty): before April 3rd, 4pm.
- Zero credit after April 3rd, 4pm.

Other Rules

- DO NOT look for any kind of help on the web outside of the Python documentation.
- DO NOT use any external libraries other than the default Python libraries and the NumPy library.
- Questions should go to Piazza first, not email. This lets any of the TAs or the instructor answer, and lets all students see the answer.
- When posting to Piazza, please check first that your question has not been answered in another thread.
- DO NOT wait until the last minute to work on the assignment. You may get stuck and last minute Piazza posts may not be answered in time.
- This is an individual assignment. DO NOT work in teams or collaborate with others. You must be the sole author of 100% of the code and writing that you turn in.
- DO NOT use code you find online or anywhere else.
- DO NOT post your code online or anywhere else.
- DO NOT turn in material you did not create.
- All cases of cheating or academic dishonesty will be dealt with according to University policy.

FAQ

a) **How will the TAs/Professor grade my HW?**
We will run your train program to learn a policy. Then we will test your dialogue system with test cases that we have created. These test cases are similar to the examples above and the examples in the lecture slides.

b) **I found a piece of code on the web. Is it ok to use it?**

No! We run plagiarism checks on the code you write. The plagiarism detector is a smart algorithm which can tell if you've copied the code from elsewhere/others.
Instead please contact TAs/Professor during the office hours with your questions.

c) **Vocareum terminates my code before it finishes, but it finishes executing on my computer.**
This usually means that your implementation is inefficient. Keep an eye out for places where you iterate. Run time issues can be solved by using efficient data structures (Hashmap, Hashset, etc.).

d) **My code works on my computer but not on Vocareum. Do I get an extension?**
The late policy above still applies, and is automatically enforced by the system. Submit your code incrementally to make sure it functions on Vocareum.

e) **When are the office hours ?**
Please refer to the course website: https://kgeorgila.github.io/teaching/cs544-spring2021/index.html and Blackboard for the Zoom links.