**Part 1b: dialog management**

Attached Files:

* [[File](https://uu.blackboard.com/bbcswebdav/pid-4607668-dt-content-rid-63502896_2/xid-63502896_2) restaurant\_info.csv](https://uu.blackboard.com/bbcswebdav/pid-4607668-dt-content-rid-63502896_2/xid-63502896_2) (10.667 KB)
* [[File](https://uu.blackboard.com/bbcswebdav/pid-4607668-dt-content-rid-63502897_2/xid-63502897_2) all\_dialogs.txt](https://uu.blackboard.com/bbcswebdav/pid-4607668-dt-content-rid-63502897_2/xid-63502897_2) (4.67 MB)

This part of the project consists of modelling and implementing a dialog management system.

**Modelling**

First, model the dialogs in the domain as a state transition diagram. All dialogs are provided in plain text as an attachment to this part of the project description. The state transition diagram is intended to provide a procedure for the system to obtain all necessary information and to include communicative utterances such as greetings. In each state the current knowledge of the system should be evaluated in order to select the information content of the next utterance, such as a question to the user, or information about a restaurant. At this point the natural language content of the utterances does not need to be considered (e.g., the information content *ask food type* is in the diagram, while the natural language content *What kind of food would you like?* is not part of the diagram). Various types of diagrams are possible, consider the examples below.

A screenshot of a diagram

Description automatically generated

The first diagram (1a) has implicit user utterances, meaning that each system utterance (represented by a rectangle) is assumed to be followed by a user utterance. In the model the expected content of the user utterance can be inferred from the following system utterance or choice node. The second diagram (1b) shows an alternative, where the user utterances are made explicit with a single node that represents all possible user utterances.  A third representation (1c) shows multiple possible user utterances followed by system responses specific to each user utterance.  
  
Note that the three diagrams are equivalent in terms of dialog states (the rectangular boxes) and state transitions (arrows). You can choose any model as long as it represents the dialog domain. The model is intended to capture the full domain, i.e., all dialogs in the dataset. Given that most dialogs are highly similar, it is sufficient to examine a limited number of dialogs (<50) in order to find almost all dialog states and transitions.  
  
The goal is to make a single diagram that covers the domain, i.e., the diagram can be used to describe (and, consequently, prescribe) all different types of dialogs in the data.  Situations that do not occur in the dataset (e.g., changing a previously stated area preference) do not need to be modelled. The diagram should be made manually, i.e., automatic extraction of state transitions is not part of the data modelling part of the project. You can use any drawing or modelling tool to produce the diagram, however handwritten diagrams are not allowed. **Number the states explicitly** so you can refer to the numbered states later in your report.

**Implementation**

Implement the diagram you have created as a dialog management system in Python. Make sure you implement the system conceptually as a **state transition function**, i.e., your code should contain a function that has the current dialog state and the current user utterance as input and the next dialog state with associated system utterance as output. Use the dialog acts resulting from your classifier from Part 1 in the state transition function. For example: at the start of the dialog, if the user utterance is classified as hello, then the dialog remains in the dialog initiation stage, and the next system utterance should be to ask the user for his preferences. Alternatively, if at the start of the dialog the user utterance is classified as inform, then the next dialog state should be to extract preferences from the inform utterance and proceed based on the number and type of preferences that is extracted.

Convert user input to lower case before classifying the dialog act (and before computing Levenshtein edit distance, see below) to match the data format used for training the classifier in Part 1a.

For extracting preferences you can use a keyword matching algorithm, i.e., if somewhere in the utterance a keyword is found that represents a type of cuisine, a location, or a price range, or if you find a typical pattern for stating a preference, you can assume that this is the user preference for the respective category. Examples:  
*I’m looking for Italian food* → *Italian* is the food type based on pattern {variable} food  
*I want to go to a bistro* → *bistro* is the food type based on the keyword list for types of food

Use Levenshtein edit distance as implemented in the python-Levenshtein library to map values to the closest domain term in case an exact match is not found. Levenshtein edit distance is defined as the number of characters that need to be inserted, deleted, or substituted (i.e., the number of *edit operations*) to convert a string into another string. As an example, consider the inform utterance “I want a restaurant serving Spenish food”. This is a misspelling of the food preference *Spanish*. According to Levenshtein edit distance, this word is close to *Danish* (substitute *S* with *D*, delete *p*, substitute *e* with *a*, total distance 3), *Polish* (also 3), *Swedish* (distance 2) and *Spanish* (distance 1). Choosing the match with the smallest distance results in the correct value *Spanish* as the food preference. If two or more words have equal distance then you can choose at random, if no word is close (distance 3 or less) then the keyword matching should fail and the system should re-ask the preference with a suitable error message. Note that there are several alternatives for the python-Levenshtein library, it is allowed to use these libraries (or copy a reference implementation directly, provided that you mention the source of the code as a comment) as long as they implement the default Levenshtein edit distance.

The system utterances can be generated based on templates, either a simple string or a pattern where variables can be inserted.

Examples:  
String:  "askpricerange": "Which price range do you prefer?"  
Pattern: "confirmfoodtype": "I did not recognize {givenfoodtype}. Did you mean {correctedfoodtype}?"

The following set of sentences can be used as test cases for preference extraction. A solution for the preference extraction part of the project is sufficient if at least these sentences are correctly handled by the system. The preferences are indicated in italics. Note that some examples express two preference types in the same sentence, such as price and location. Some examples show a *don't care* preference, such as 7.

1. I'm looking for *world* food
2. I want a restaurant that serves *world* food
3. I want a restaurant serving *Swedish* food
4. I'm looking for a restaurant in the *center*
5. I would like a *cheap* restaurant in the *west* part of town
6. I'm looking for a *moderately* priced restaurant in the *west* part of town
7. I'm looking for a restaurant in *any* area that serves *Tuscan* food
8. Can I have an *expensive* restaurant
9. I'm looking for an *expensive* restaurant and it should serve *international* food
10. I need a *Cuban* restaurant that is *moderately* priced
11. I'm looking for a *moderately* priced restaurant with *Catalan* food
12. What is a *cheap* restaurant in the *south* part of town
13. What about *Chinese* food
14. I wanna find a *cheap* restaurant
15. I'm looking for *Persian* food please
16. Find a *Cuban* restaurant in the *center*

When all preferences are extracted, the system needs to provide a restaurant recommendation to the user. Write a Python function to retrieve restaurants from a CSV database (see the file as attached to this part of the project description) according to the preferences. If there is no restaurant that conforms to the requirements then the system should inform the user that there is no restaurant that conforms to the requirements of the user. If there are multiple possible restaurants then the system should select one of these at random and store the remaining matches in case the user asks for an alternative suggestion using the same preferences. Take into account that the input is converted to lower case when implementing the lookup function.

**Deliverables**

* The state transition diagram in graphical form with numbered states
* A working dialog system interface that prints system utterances to the screen and processes user input utterances entered with the keyboard, **implementing a state transition function**corresponding to the diagram, using predictions from the classifier built in Part 1a as part of the input for state transitions
* An algorithm identifying user preference statements in the sentences using pattern matching on variable keywords and value keywords on utterances classified as inform, using Levenshtein edit distance if necessary
* A lookup function that retrieves suitable restaurant suggestions from the CSV database matching the preferences as extracted in the implemented algorithm