DCUBuddy
Technical Specification

Student 1 Name: Mark Kevin Queypo
ID: 19404214

Student 2 Name: Conor Patrick Marsh
ID: 19728351

Date: 4/3/22

0. Table of Contents

1. Overview

1.1 Introduction

The DCUBuddy presents a chatbot which answers text-based queries from the user. The chatbot provides useful campus-related information including information on assignments and timetables. The ultimate aim is to provide an alternative platform to the current Loop[1] system, by presenting some of the more dynamic information requirements that a student would need at short notice.

DCUBuddy's main target user would be a first-year student of DCU who may be unfamiliar with certain fundamental aspects of campus navigation. Users can access information on assignments, timetable and campus maps. There is a certain amount of general information which can also be queried.

1.2 Motivation

DCUBuddy is primarily motivated by the designers' ambitions to improve on the basic Loop design and its lack of any live interaction with the student. It is noteworthy that certain features such as adjustable assignment information is not provided on Loop. Locating campus maps through Loop presents difficulties which can be improved upon. There is a prevalence of formalised information in the Loop design which the designers of DCUBuddy felt could be replaced with more relevant information provided in a friendlier, more informal manner.

2. High-Level Design

This section of the document describes each component and how they interact with each other to create DCUBuddy.

[1] loop.dcu.ie

2.1 Context Diagram

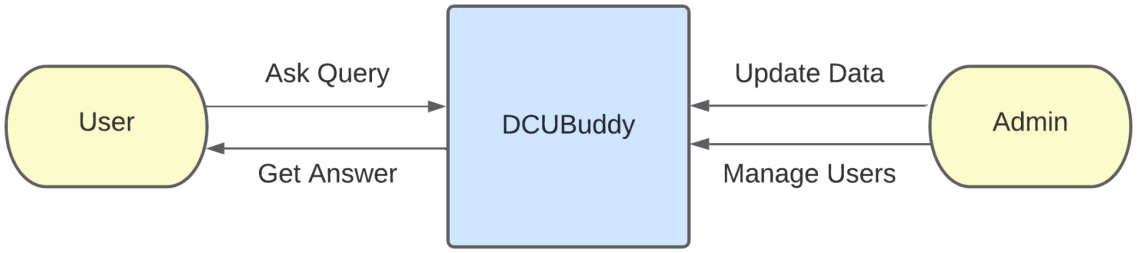


Figure 1: Context diagram of DCUBuddy

The context diagram in Figure 1 gives a brief summary of how users can interact with DCUBuddy. To clarify the diagram, “User” refers to the category of people who are using the chatterbot on DCUBuddy without any responsibility of the system and “Admin” refers to the category of people who have responsibilities of managing the system.

2.2 Data Flow Diagram: Query Handling

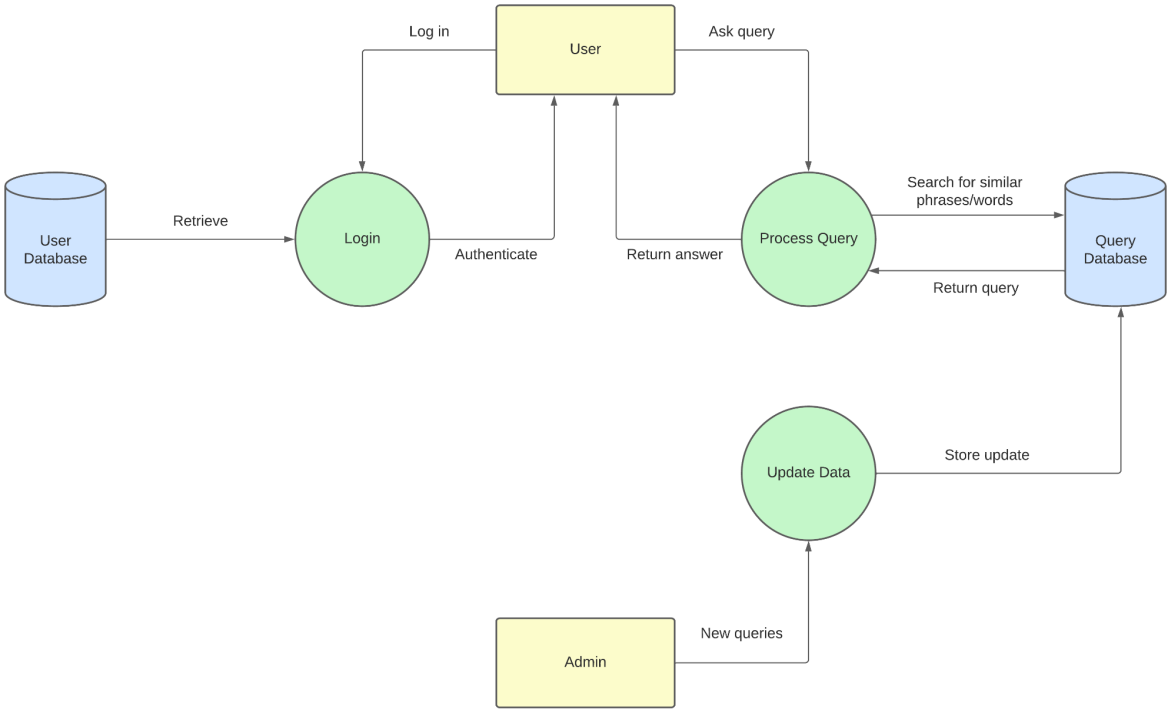


Figure 2: Data flow diagram of DCUBuddy Query Handling

Figure 2 above shows how users interact with the web application through the UI. It shows how the chatterbot processes a user’s query. It also shows how administrators update data to the database.

2.2.1 Login

Users before accessing the main chatbot page would need to log in or sign up for a new account. Details required are email, password and student's course code.

2.2.2 Processing Query

The chatterbot takes a user's query and searches using an algorithm explained in Section 4.1.1 that searches for similar words and phrases from the query database. If successful, the chatterbot will return the appropriate response to the user. If not successful, the chatterbot will send a message to the user that it did not find an appropriate response to their query.

2.2.3 Updating Query Database

The administrators of DCUBuddy can update or add new data to the chatterbot's query database. If successful, the chatterbot will be able to respond better to existing queries or learn new responses to newer queries. More details on how admins can update data in Section 4.1.3.

2.3 Data Flow Diagram: Fetching Timetable

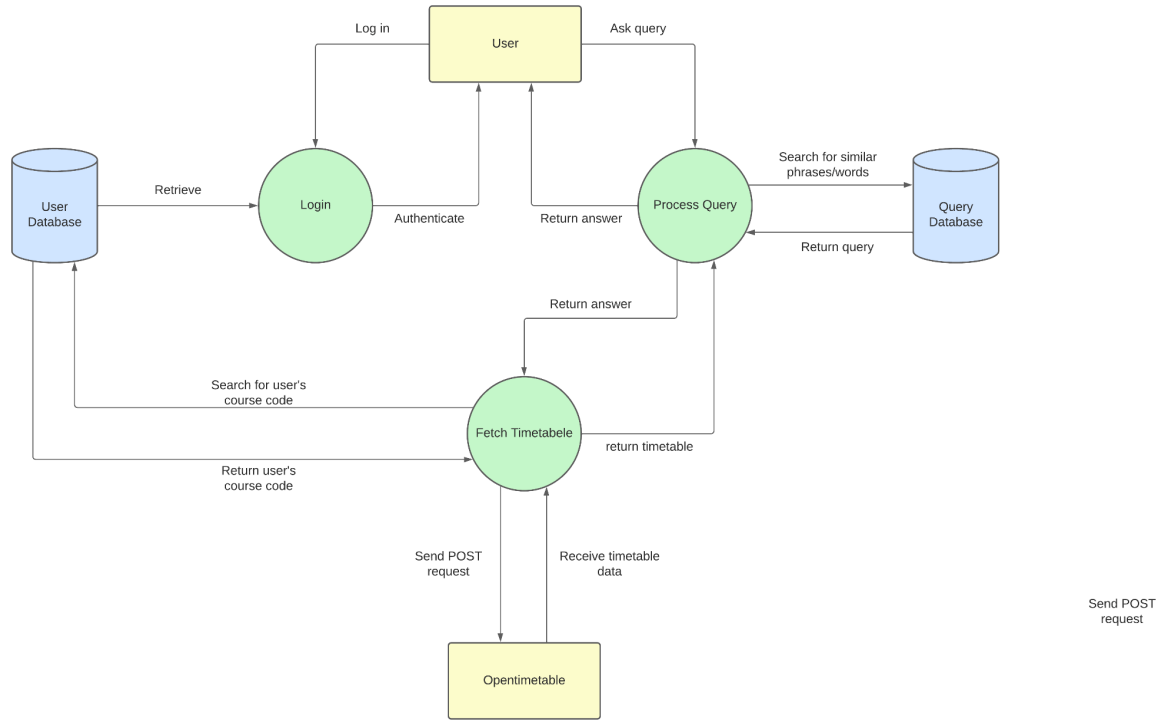


Figure 3: Data flow diagram of fetching student timetables from opentimetable.dcu.ie

Figure 3 above shows how the user can ask the chatterbot for their student timetable and how a system returns the appropriate timetable.

2.3.1 Fetching User's Course Code

When a user asks for a student timetable, the system will first find out what course timetable they should send. It will check the user database and find out the current user's course code associated with the account.

2.3.2 Fetching Timetable

Once it gets a user's course code, it will send a POST request[2] to the opentimetables [3] website to fetch the user's student timetable in respect to their course code. The broker api for opentimetables is public and retrieving data from the server is easy.

```
def request_events(course_code, data):
    """
        Getting a response from website
    """
    res = requests.post("https://opentimetable.dcu.ie/broker/api/categoryTypes/241e4d36-60e0-49f8-b27e-99416745d98d/categories/events/filter", json=data, headers=HEADERS)
    if res.status_code != 200:
        logging.critical("Unable to get request for course with code: %s", course_code)
        return("Unable to access timetable.")
    else:
        logging.debug("Succesfully got request for course with code: %s", course_code)
        result = json.loads(res.text)
        ongoing = result[0]['CategoryEvents']
        return ongoing
```

Figure 4: Sending a POST request to the broker api of opentimetables

2.4 Data Flow Diagram: Assignment Manager

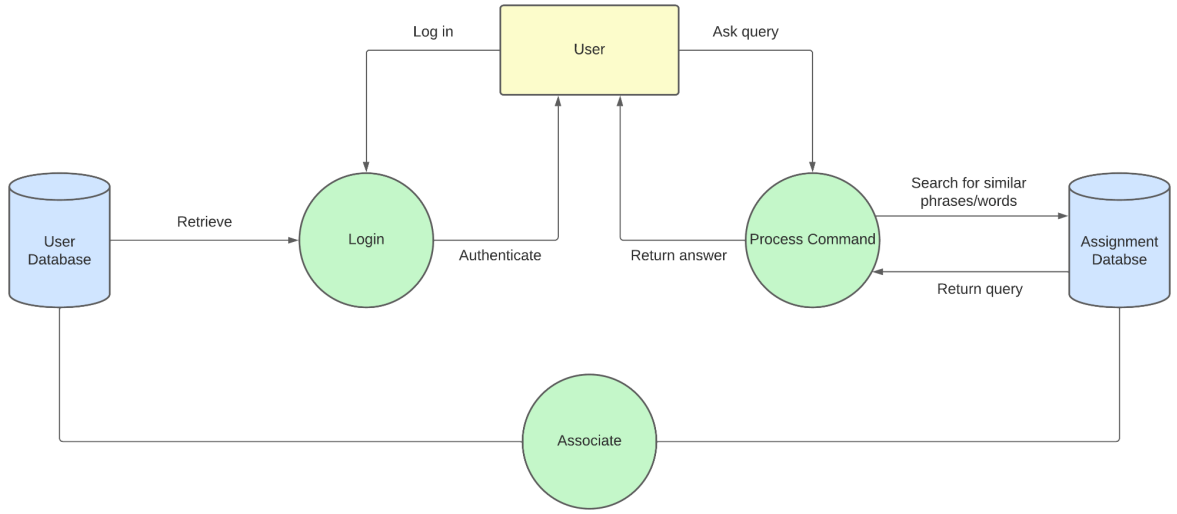


Figure 5: Data flow diagram of assignment management

[2] POST request: A request method supported by HTTP, enclosed data is sent to the server as a request message.
[3] opentimetable.dcu.ie

2.4.1 Process Commands

Chatterbot has a set of commands that act as tools. In this case, it is the assignment management feature. For a list of commands and their functionality, please refer to the user manual for DCUBuddy.

2.4.2 Account and Assignment Relationship

Each assignment is associated with one account only so users can only see the assignments they have added.

3. System Architecture

2.1 Dependencies

Languages

- Python
    - Version: 3.7 or below (Compatible without extra steps).
    - Version: 3.8 and above (requires changes to `sqlalchemy` module - see section 2.2).
    - Used by: Backend
- Javascript
- HTML

Databases

- SQLite
    - Version: 3.35.0
    - Used by: Account and Assignment Management, Chatterbot.

Python Modules:

- Flask[4]
    - Version: 2.0+
- SQLalchemy[5]
    - Version: 1.2.19

[4] https://readthedocs.org/projects/flask/
[5] https://docs.sqlalchemy.org/en/14/

- ChatterBot[6]
    - Version: 1.0.4
- Chatterbot_corpus[7]
    - Version: 1.2.0
- PyYAML[8]
    - Version: 5.1.2
- Requests[9]
    - Version: 2.27.1
- WTForms[10]
    - Version: 3.0.1

`Flask` is a web framework in the form of a Python module for developing web applications. Flask uses the Web Server Gateway Interface (WSGI) as a standard for Python web application development. Werkzeug is a WSGI toolkit within Flask that implements requests, response objects, and utility functions. **`jinja2`** in Flask's template engine which combines with a specific data source to render a dynamic web page.

`SQLalchemy` is a Python SQL toolkit and Object Relational Mapper. It is a library that facilitates the communication between Python programs and databases.

`ChatterBot` is a machine learning engine which makes it possible to generate intelligent chat responses using training data from previously saved conversations. The accuracy of ChatterBot's responses increases with the amount of input it receives from the user. It can select the closest matching known statement that matches the input.

`Chatterbot_corpus` is a machine readable multilingual dialogue corpus. The module is used to quickly train ChatterBot to respond to various inputs in different languages. Personalised training data can be added using **`.yml`** files.
PyYAML is a python module for handling YAML - a data serialisation format designed for human readability and interaction with scripting languages. It is used to provide training data for the chatbot.

---

[6] https://buildmedia.readthedocs.org/media/pdf/chatterbot/latest/chatterbot.pdf
[7] https://chatterbot-corpus.readthedocs.io/_/downloads/en/latest/pdf/
[8] https://pyyaml.org/wiki/PyYAMLDocumentation
[9] https://docs.python-requests.org/en/latest/
[10] https://wtforms.readthedocs.io/en/3.0.x/

```yaml
categories:
- greetings

conversations:
- - Hi
  - Hello, welcome to DCUBuddy

- - who are you?
  - i am a chatbot named DCUBuddy

- - Hello
  - Hi, I am DCUBuddy
```

Figure 6: greetings.yml which has data on queries

`Requests` is a HTTP library. Requests allows you to send HTTP/1.1 requests extremely easily.

`WTForms` integrates with the Flask framework to provide data validation when submitting data through the program's login forms.

4. Implementation

This section will explain how each component that makes up the chatbot works and how each it is implemented.

4.1 Chatterbot

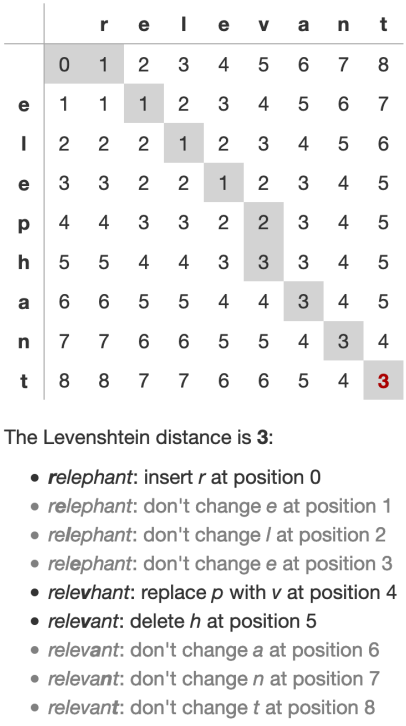
The chatbot's ability to talk intelligently is based on efficient retrieval of potential candidate statements with which to issue a response. In searching for the desired response the chatbot makes calculations such as the following:

- What is the similarity of the user's statement to known statements?
- What is the frequency with which these known statements occur?
- How likely is the user's statement to fit into a given category of statements?

4.1.1 Comparisons

There are a number of statement comparison functions built into Chatterbot which are based on particular algorithms. Examples include use of the Jaccard index and

Levenshtein distance. The Jaccard Index compares one sample set with another to identify which elements are shared and which are distinct. The Levenshtein distance essentially represents the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into another. Alternatively, a user-designed comparison algorithm can be coded in. The statement_comparison_function parameter sets the statement comparison method for the chatbot. The chatterbot is currently using the Levenshtein algorithm for comparisons. The algorithm finds the Levenshtein distance between two statements which are the number of single-character edits which can be done to a word which include insertions, deletions and substitutions required to modify one word into the other word.



		r	e	l	e	v	a	n	t
	0	1	2	3	4	5	6	7	8
e	1	1	1	2	3	4	5	6	7
l	2	2	2	1	2	3	4	5	6
e	3	3	2	2	1	2	3	4	5
p	4	4	3	3	2	2	3	4	5
h	5	5	4	4	3	3	3	4	5
a	6	6	5	5	4	4	3	4	5
n	7	7	6	6	5	5	4	3	4
t	8	8	7	7	6	6	5	4	**3**

Figure 7: Example of calculation of the Levenshtein distance from the word "elephant to the word "relevant" extracted from https://phiresky.github.io/levenshtein-demo/

4.1.2 Preprocessors

The preprocessors are simple functions that the chatterbot can use to modify any input the chatterbot receives before a statement gets processed by the logic adapter. The preprocessors used for this project are:

- **`clean_whitespace`:** This removes any consecutive white spaces from input given.

- **`Unescape_html:`** This converts escaped characters into unescaped html characters. i.e “<b>” becomes “<b>”.

4.1.3 Logic Adapters

Logic adapters are used by Chatterbot to select a response to an input statement. There are two main steps which the logic adapter performs. Firstly, the database is searched for a statement that matches or closely matches the input statement. Secondly, a known response to the given input statement is identified. Often several known responses will be available for the given statement.

The logic adapter used for DCUBuddy is the BestMatch logic adapter which selects a response based on the best known match to a given statement. It will then return a response statement that has the highest confident score.

An “**`input_statement`**” is a statement that closely matches an input to the chatbot. A **“`response_list`”** is a list of options to choose a response from. The response selection is based on the response selection method, in DCUBuddy we are using **“`get_most_frequent_response`”**. This will return the response statement with the greatest amount of occurrences. Alternatives would be to choose the first statement, a random statement, or to create one’s own algorithm for choosing a response. The response selection method is passed to the program upon initialization.

Figure 8 shows how the process works. The input text is modified by the preprocessors which is then handed to the logic adapter which finds similar statements and grades them by confidence points. The response with the highest confidence points is used and returned by the chatbot.

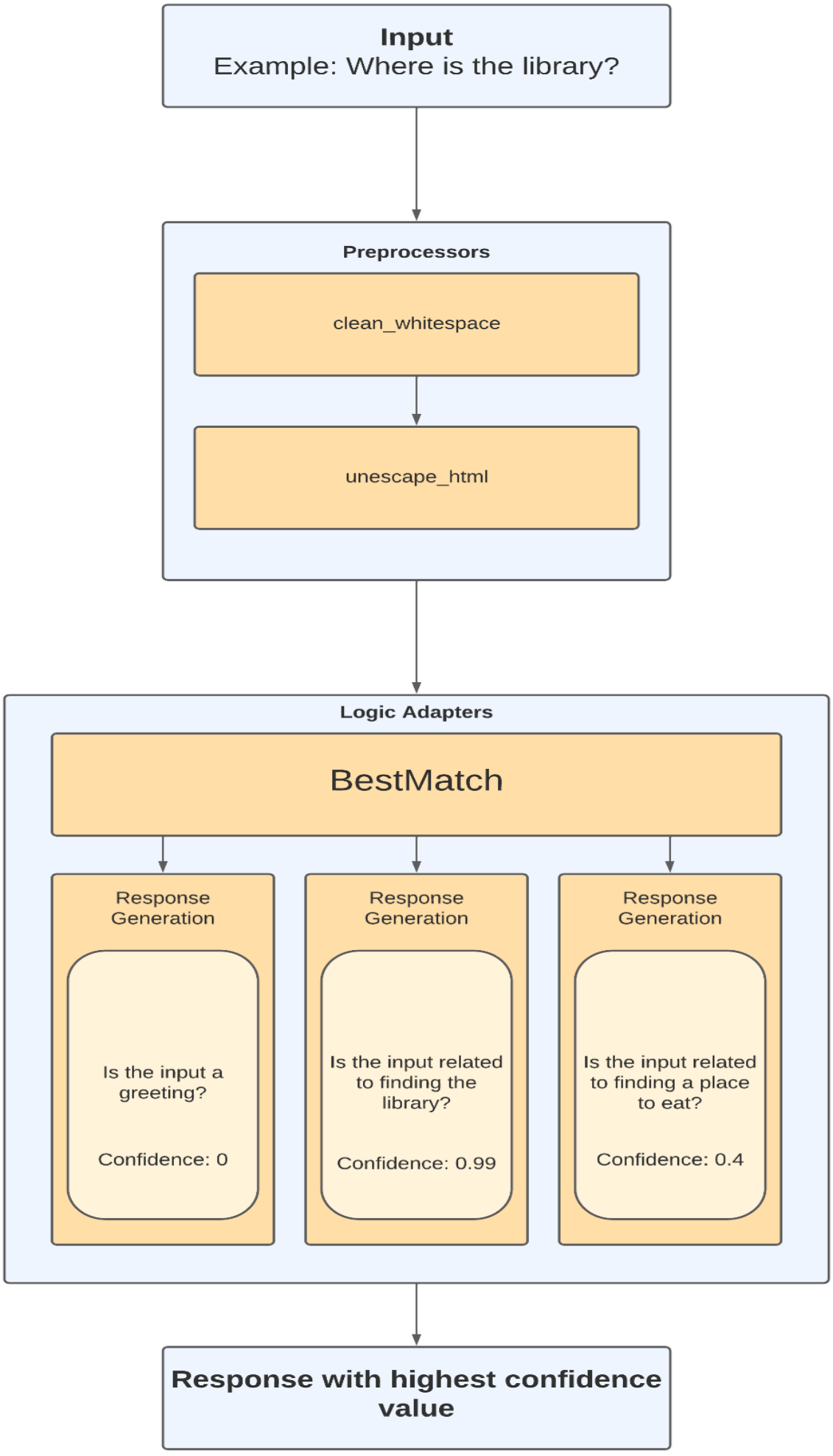

Input
Example: Where is the library?
Preprocessors
clean_whitespace
unescape_html
Logic Adapters
BestMatch
Response Generation
Is the input a greeting?
Confidence: 0
Response Generation
Is the input related to finding the library?
Confidence: 0.99
Response Generation
Is the input related to finding a place to eat?
Confidence: 0.4
Response with highest confidence value

Figure 8: An example of how a response is processed with preprocessors and the logic adapter

4.1.4 Chatterbot Corpus

The ChatterBot Corpus is a project containing user-contributed dialog data that can be used to train chatbots to communicate. It is a sample of various input statements and their responses. DCUBuddy's training involves loading example dialog data into the chatbot's database. The creates a data structure of known statements and responses or builds upon the existing structure. A chatbot trainer is provided with a data set and represents the statement inputs and responses in a knowledge graph. A knowledge graph illustrates the relationship between a number of objects or events, representing these in a database and as a graph structure (see Figure 9 below). The training graphs are the known statements and responses given to the chatbot to establish a conversation.For the training process, an example dialog from the chat's database is loaded. The graph data structure is created or built upon the known statements and responses. The trainer is called CorpusTrainer. The trainer is given a data set where a data set where it thens creates entries to the knowledge graph which represents the inputs and responses. For finding the training data, refer to the training_data folder.

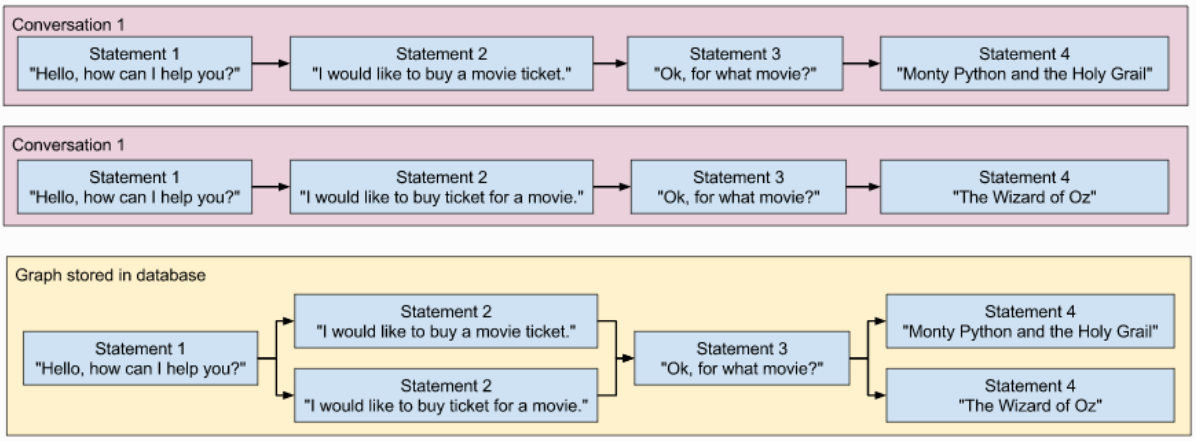


Figure 9: An example of a knowledge graph from chatterbot docs [Link]

4.1.5 Updating Data

An admin can add new queries by adding statements to the training data files. Different training data files exist for different aspects of conversation such as greetings, general commands, map, assignment and timetable information. Statements must be in a .yml format in order to be interpreted correctly. Each item in the training data can be established as a possible response to its predecessor in the file.

5. Problems Solved

5.1 Using ListTrainer

Before we used the corpus trainer to train our chatterbot, we have used the other trainer, the ListTrainer. Unlike `Corpustrainer`, `ListTrainer` takes from a list instead of from a yml file. So we made different text files, split each line and added them together to feed to the bot. After more research, we discovered that this method of training the chatterbot is wrong. What happens is, when you give the chatterbot a list, it will treat that whole list as a whole entire conversation. So when you enter an input it, it will respond with the next element in the list. However, if you send an input that is further on on the list or is an element earlier on the list, the chatterbot will not be able to understand the statement and find any appropriate response.

The problem was fixed, when we switched to using the `CorpusTrainer` instead which handled yml files instead. Compared to the text files, the yml files are more organised and easier to understand where you are able to name a category to the set of queries. Conversations can also be easily nested into the files which offers more ongoing conversations with the chatbot.
Figure 10 shows an example of training data inside a yml file.

```yaml
categories:
- greetings

conversations:
- - Hi
  - Hello, welcome to DCUBuddy

- - who are you?
  - i am a chatbot named DCUBuddy

- - Hello
  - Hi, I am DCUBuddy

- - what can you do?
  - You can ask me anything to help you with your time in DCU. I also have special commands that can help y
    Do you want to see?
  - sure
  - To add assignments <br> !addassignment [Assignment_Name] [Due_Date] <br><br> To delete assignments <br>

- - what can you do?
  - You can ask me anything to help you with your time in DCU. I also have special commands that can help y
    Do you want to see?
  - "yes"
  - To add assignments <br> !addassignment [Assignment_Name] [Due_Date] <br><br> To delete assignments <br>

```

Figure 10: Example of data in greetings.yml

6. Testing

6.1 Strategy

We decided to adapt an agile approach when it comes to the development process of DCUBuddy. Time is limited during the development of the project as we have to make sure we reach deadlines and key dates, so using a more formal approach for the project would be difficult. Due to the time constraints, we are not able to correct any design mistakes. So development of this project involved interactive development and testing. The project undergoes a process of developing and testing with repeated tweaking such as making small changes or adding new features and testing through different stages of development. The aim is to remove or reduce any issues that affect usability and user experience ahead of the deadline and when we did user testing.Of course, this means that not all elements of the system are exhaustively tested due to the aforementioned time constraints.

6.2 User Testing

The primary testing mechanism that was used for our project was user testing. This testing was fully aligned with the ethical conditions required by DCU in terms of confirming full consent of test volunteers through the necessary procedures. The testing itself was undertaken using the SUS (System Usability Scale). Five volunteers were recruited to, where possible, try the system hands-on while being observed and guided by one of the designers. Failing this, a comprehensive video demonstration[11] was made available to the testers in order for them to experience the system. Following this, a survey[12] of ten questions was provided with questions answered on a scale of 1 to 5 reflecting level of agreement with certain statements. Each statement acted as an inversion of the previous one in terms of positive versus negative sentiment towards the system. This helped to ensure consistency of opinion and helped ascertain that ratings were not being randomly chosen by the tester.

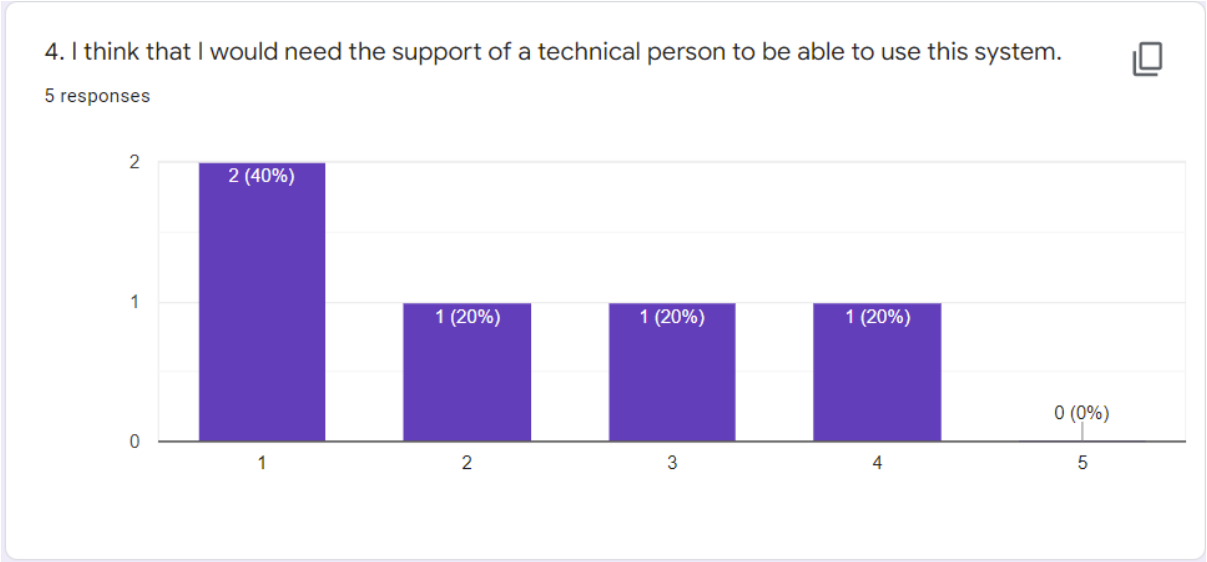
Test results included the following SUS total scores (rated from 0 to 100 with 100 being the highest level of satisfaction from the user):

User 1 - 77.5
User 2 - 85
User 3 - 90
User 4 - 100
User 5 - 100

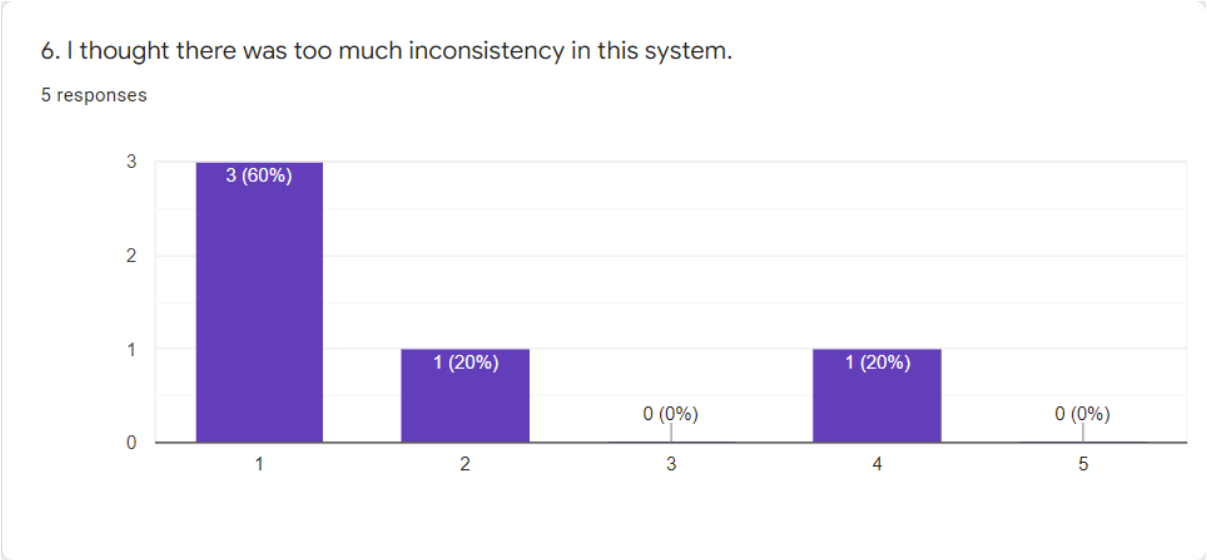
[11] https://youtu.be/6Sy7Z_eEoYs
[12] https://docs.google.com/forms/d/e/1FAIpQLSeFpYbbYZyqJdh0rbvzvpAgICHb3Zu3ceDF7BPoPFqoL1ZiBw/viewform?usp=sf_link

Generally high scoring overall, with an average SUS score of 90.5, reflected a very positive level of satisfaction with the system experience.

Certain findings which to some extent differed from this positive feedback included the following:



Mixed ratings for technical understanding of the system showed that users found it difficult to grasp how the system worked of their own accord.



A degree of inconsistency in the system was identified by certain testers.

Additional suggestions provided by the testers included the idea of enabling access to a lecturer's timetable in order to inform as to when they would be free for

appointments. Also suggested was the inclusion of a “Did you mean…?” feature for clarification of queries which were not well understood by the chatbot.

6.3 Unit Testing

Chatterbot has inbuilt tests that can be run using the nose[13] module. Nose can be installed using pip. When running unit tests, just run the command:

```
nosetests [Path_to_test]
```

We use a keyword to see if the chatterbot finds matches. If it manages to find matches to the key phrases, then it has passed thes test. This also makes sure that functions work properly and that we are getting no errors.

```python
from tests.base_case import ChatBotTestCase
from chatterbot.trainers import ChatterBotCorpusTrainer


class ChatterBotCorpusTrainingTestCase(ChatBotTestCase):
    """
    Test case for training with data from the ChatterBot Corpus.
    Note: This class has a mirror tests_django/integration_tests/
    """

    def setUp(self):
        super().setUp()
        self.trainer = ChatterBotCorpusTrainer(
            self.chatbot,
            show_training_progress=False
        )

    def test_train_with_english_greeting_corpus(self):
        self.trainer.train('chatterbot.corpus.english.greetings')

        results = list(self.chatbot.storage.filter(text='Hello'))

        self.assertGreater(len(results), 1)

    def test_train_with_english_greeting_corpus_search_text(self):
        self.trainer.train('chatterbot.corpus.english.greetings')

        results = list(self.chatbot.storage.filter(text='Hello'))

        self.assertGreater(len(results), 1)
        self.assertEqual(results[0].search_text, 'hello')
```

[13] https://nose.readthedocs.io/en/latest/

7. Future Work

7.1 Machine Learning

Machine Learning possibilities exist to improve the informativeness of answers, better the maintenance of the context of the dialogue, and present a more human, helpful or interesting personality through the bot. More training data and additional samples of training data will lead to improvements in overall performance. Other possibilities include the adaptation of an NER (Named Entity Recognition) model to find different entities that are present in the text such as persons, dates, organisations, and locations. Potential capabilities in recognising persons such as lecturers within college would tie in with the user testing suggestions, specifically enabling the chatbot to provide timetable information regarding the lecturer's availability at a given time. Information on entities such as societies could be improved with name recognition of individual societies. Recognition of a date could lead to a more direct way of providing relevant information related to that date (e.g. timetable, information on temporary service shut downs, society or college events on that date). Specific location information would be a clearly useful feature as the potential exits for location details of specific campus buildings or rooms to be provided.