Interactive Natural Language Processing Based Artificial Intelligence System

An Interactive NLP-Based AI Chatbot System for COMP3074: Human-AI Interaction

CHUTIYADA CHEEPCHIEWCHARNCHAI, School of Computer Science, University of Nottingham, UK

Additional Key Words and Phrases: Natural Language Processing, Chatbot Architecture, Conversational Design

1 Introduction

This paper presents the architecture and conversational design of an NLP-based chatbot system. The system integrates six key functionalities: intent matching, identify management, transaction (transportation reservation), information retrieval & question answering, small talk, and sentiment analysis. The system also incorporates design principles, including prompt design, discoverability, error handling, personalisation, confirmation, and context tracking. The evaluation and discussion sections of the paper provide reflections on the system's performance, and on the project as a whole.

2 Chatbot Architecture

The chatbot architecture describes the structural design of the system. It defines how the system tasks function, along with their implementations and importance to the system.

2.1 System Structure

```
HAI_CW_20421241/
 ├─ classifier/
 │   ├─ train_sentiment_classifier.joblib
 │   └─ train_sentiment_transformer.joblib
 ├─ data/
 │   ├─ sentiment/
 │   │   ├─ negative/
 │   │   │   └─ ...txt
 │   │   └─ positive/
 │   │       └─ ...txt
 │   ├─ ask_name.csv
 │   ├─ badwords.csv
 │   ├─ booking.csv
 │   ├─ database.db
 │   ├─ QA_dataset.csv
 │   ├─ small_talk.csv
 │   └─ train_sentiment.csv
 ├─ data_processed/
 │   ├─ ask_name_matrix.joblib
 │   ├─ ask_name_preprocess.joblib
 │   └─ ...
 ├─ util/
 │   ├─ util_classifier.py
 │   ├─ util_func.py
 │   └─ util_text_process.py
 ├─ booking.py
 ├─ discover.py
 ├─ farewell.py
 ├─ identity_management.py
 ├─ main.py
 ├─ QA.py
 ├─ reservation_management.py
 └─ small_talk.py
```

Fig. 1. The Top-down layout of the source code.

Author's Contact Information: Chutiyada Cheepchiewcharnchai, psycc6@nottingham.ac.uk, School of Computer Science, University of Nottingham, UK.

Fig. 1 shows the structure of the source code. This source includes the "`data`" folder, which holds the datasets and the database required for the system to function, the "`data_processed`" folder, which stores the processed matrices and transformers of the datasets, and the "`classifier`" folder, which stores the transformer and classifier related to the sentiment analysis. Storing extra processed data like this allowed a significant decrease in the system's response time, as the whole dataset is not being processed every time an input is placed.

As for executable codes, the system runs using "`main.py`". Utility codes in the "`util`" folder are used to implement functions that are needed in multiple files. All other "`.py`" files belong to independent architectural functions within the system, divided for easier file management, apart from "`reservation_management.py`", which will be described further in section 2.4.2.

2.2 Intent Matching

2.2.1 Functionality

Intent matching is used by the chatbot system to identify the objective of each user's input, in order to provide an appropriate outcome or action for each input. This task is one of the most important in a chatbot system. Without intent matching, the system cannot differentiate how to respond to more than one type of input, preventing the system from being able to run more than one task.

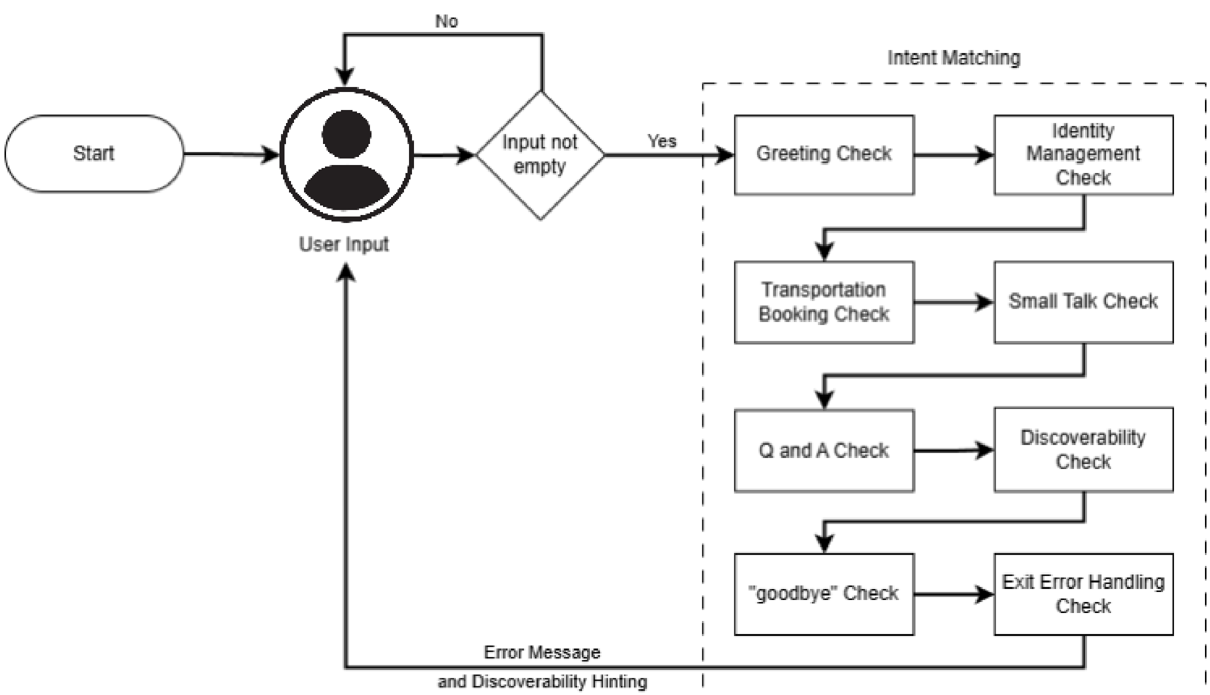

Fig. 2. The flow diagram of main.py, illustrating the workflow order of the intent matching in the system.

2.2.2 Implementation & Justification

The system's intent matching follows a workflow shown in Fig. 2, and is implemented in "`main.py`". It is implemented by validating each "`check`" function, one by one.

Two types of intent matching algorithms were implemented in the system. Cosine similarity matching [3] with TF-IDF (Term Frequency-Inverse Document Frequency) weighting was used for larger intent matching datasets, such as question answering and small talk. The algorithm uses a preprocessing technique which includes tokenisation, word tagging, lemmatisation, and removal of all stop words and punctuations. This is important when dealing with a large dataset, as reducing the amount of tokens can significantly improve the speed of the system's processing and response time. The mathematical implementation of this algorithm is further discussed in section 2.5.2.

The other implementation deals with smaller datasets, specifically datasets with shorter sentences, and fewer key terms that can be used to identify it as a certain intent. This algorithm includes a smaller list of stop words to be removed, and bases an input's intent matching score solely on the ratio of tokens it has against a list of tokens given to the algorithm. This ratio is compared against a given threshold to ensure the ratio is high enough for an input to be considered a specific intent. The reason for implementing this second algorithm is that the previous algorithm removes a lot more tokens, which ends up removing whole inputs, or leaving so few words that it can no longer be used to accurately predict intent. The second algorithm, therefore, permits a scaled-down preprocessing alternative which can maintain the meanings of smaller inputs and datasets. See Appendix A.1.1 and A.1.2 for both algorithm implementations.

Both algorithms also take in threshold values, which are used to determine whether the similarity scores returned by the algorithms were high enough to match the intent correctly.

2.3 Identify Management

2.3.1 *Functionality*

Identity management is used to identify and keep track of the user's identity, and is implemented in "`identity_management.py`". The system does this by storing the user's name in a variable in "`main.py`". This gives personalisation to the system, and can also be used to streamline other functionalities, such as automatically inputting names into transport reservations.

2.3.2 *Implementation & Justification*

The three types of inputs that the identity management can identify are: greetings, query for the user's name, and query to change name. In all three cases, if the user hasn't already given the chatbot a name, it will prompt the user to do so. This is so that there are multiple opportunities for users to be prompted for their name, even if they did not know that the system has this function, improving function discoverability.

The name can be entered independently, but when given as part of a sentence - such as "I want to change my name to Gabi." - a tagger is used to identify the noun of the sentence, collecting it as the name. The decision to use a tagger is to allow more dynamic inputs, enabling users to enter their needs all in one go, reducing completion time, and improving user experience.

A list of inappropriate words was also generated (see "`data/badwords.csv`") to prevent certain names from being used. This helps avoid offensive content, and keep the usages of the system more professional.

2.4 Transaction: Transportation Reservation

2.4.1 *Functionality*

The chatbot system's main function is the transportation reservation function, and is implemented in "`booking.py`". It has two main tasks within the function: creating and storing new reservations, and viewing old reservations. The user's name, transport type, date of travel, time of travel, and email are required to make a reservation, and only the user's name and the auto-generated reservation ID are required to view a reservation.

2.4.2 *Implementation & Justification*

Each task within the transaction operates in its own while loop, which can be entered by inputting commands such as "I want to book a reservation", or "view booking". Using a loop allows the system to focus only on commands about reservation, increasing its efficiency and accuracy.

For both reservation tasks, the while loop keeps track of the data that are given, prompting the user for missing information. Error handling is used in the while loops, reminding users to ensure correct data and format are being entered. Appendix A.2.1 and Appendix A.2.2 shows the full run-through of system transactions.

Since a reservation requires multiple different data of different types, a helper class called "`Reservation`" in "`reservation_management.py`" is implemented to parse and check data validity, and store them appropriately. This class has functions to set data and get the completion stage of reservations, allowing for easier tracking of data. The while loops in "`booking.py`" also became less complex to implement, as only one object needed to be retained. Appendix A.2.3 illustrates the "`Reservation`" class diagram.

Another feature of the reservation tasks is the ability to store and retrieve data from "`database.db`", through using "`sqlite3`". This allows the system to retain reservation data from previous runs, ensuring data is not lost when the system is exited.

The input commands for both tasks are also dynamically handled, allowing users to input extra information into one command, in order to reduce the number of interactions. This is explained further in section 3.3.3.

2.5 Information Retrieval & Question Answering

2.5.1 *Functionality*

Information retrieval is a technique used to find and recover information from a given query. For this chatbot system, the dataset "`QA_datatset.csv`" is used to query random factual information for users. Most of the information retrieval mechanics are implemented in the "`util/util_text_process.py`", but then called into "`QA.py`" to create a high-level implementation for easier usage.

2.5.2 *Implementation & Justification*

The chatbot system uses cosine similarity and TF-IDF weighting to calculate how closely a user's input is to a query in the system's dataset. To achieve this, the Bag-of-Words model is firstly used to represent the user's query. As discussed by Jain et al. [1], this representation ignores the order of words, only considering " their presence or absence in the document". However, this led to an advantage for longer documents, which held more words, giving them higher similarity scores, despite not necessarily being contextually more similar. Therefore, as demonstrated in Equation 1 and Equation 2, the *log* function using TF-IDF can dampen the effect of differing document lengths. Furthermore, while Equation 1 is used to increase the score when there are similar terms between documents, Equation 2 is used to further dampen popular words, as it converges weights of frequent terms toward $log(1)$ or 0. The equations are placed together into Equation 3, which reduces the importance of frequent terms, and effects of document lengths.

$$\mathrm{TF}(t,d) = \log\left(1 + freq(t,d)\right) \tag{1}$$

$$\mathrm{IDF}(t) = \log\left(\frac{N+1}{n+1}\right) + 1 \tag{2}$$

$$\text{TF-IDF}(t,d) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t) \tag{3}$$

Variable Definitions:

- t = Term in the document.
- d = Document in the corpus.
- $f(t,d)$ = Raw frequency of term t in document d.
- N = Total number of documents in the corpus.

- n = Number of documents in the corpus that contain the term t.

The cosine similarity is then applied to the vectorised TF-IDF representation of the documents, to calculate the cosine value of the angular differences between two documents, as shown in Equation 4. The benefit of using cosine similarity is that calculating the angle between documents makes the magnitudes of the vectors obsolete, which is particularly great when dealing with high-dimensional data, or in this case, dealing with large amounts of documents with many terms.

$$\text{cosine-similarity}(\mathbf{d_1}, \mathbf{d_2}) = \frac{|\mathbf{d_1} \cdot \mathbf{d_2}|}{\|\mathbf{d_1}\|\|\mathbf{d_2}\|} \tag{4}$$

After the similarity score is calculated, all queries with the top scores are randomised to return one corresponding answer to the user each time, allowing random variations between each information retrieval job.

2.6 Small Talk

2.6.1 Functionality

Small talk refers to the ability to have short, non-consequential interactions with the conversational system. This was implemented in “`small_talk.py`”, where three different tasks can be performed: responding to a query about the chatbot’s well-being, extracting weather information from the Met Office website, and responding to gratitude.

2.6.2 Implementation & Justification

Similarly to information retrieval, small talk uses cosine similarity to calculate how closely related an input is to the query in its dataset (`small_talk.csv`). However, instead of querying for answers, the small talk dataset returns the *type* of small talk it could be. The reason for this is that returning just the type of small talk allows for more flexibility in responses. Instead of rigid answers, which are important for returning facts in information retrieval, the system promotes variety in small talk by having a pool of answer templates for each type of small talk task, which then randomly gets returned to users. This creates a less predictable and more engaging system for the user.

To retrieve the current day’s weather, the system uses “`BeautifulSoup`” to parse the URL “https://weather.metoffice.gov.uk/forecast/gcrjm8jf7” for the highest and lowest daily temperature, returning them to the user. The system also added a comment on whether it thought the temperatures were considered warm, cold, or just right. Fig. 3 shows how “`BeautifulSoup`” parses the Met Office website data, and returns the daily temperature to the system. Since the website is always updated to the latest daily weather, using this implementation means the data aren’t hard-coded, and will also dynamically update each day.

2.7 Sentiment Analysis

2.7.1 Functionality

Sentiment analysis is the use of language processing and machine learning to understand and classify the tone of a text. Although not greatly explored in this project, sentiment analysis was implemented to small talk, in order to classify the tone of the user’s response to the question "How about yourself?", in the context of asking about the user’s well-being. The sentiment analysis can classify the user’s response as either positive or negative, with any low-level confidence response being considered a neutral sentiment.

2.7.2 Implementation & Justification

The sentiment analysis was created through experimenting with four different types of classifiers, including Logistic Regression, MNB (Multinomial Naive Bayes), SVM (Support Vector Machine), and Decision Tree. See

```
# scrape weather temperatures from the Met Office website
1 usage
def readWeather():
    url = "https://www.metoffice.gov.uk/weather/forecast/gcrjm8jf7"
    content = BeautifulSoup(request.urlopen(url).read(), features: 'html.parser').get_text()

    textContent = []
    for line in content.splitlines():
        if line.strip() != '':
            textContent.append(line.strip())
    i = textContent.index("Today")
    temps = textContent[i+1].split("   ")

    return (temps[1].split(" ")[0]), (temps[0].split(" ")[0])
```

Fig. 3. The implementation of BeautifulSoup to parse Nottingham's daily temperatures back to the system.

Appendix A.3.1 for experiment implementation and Appendix A.3.2 for experiment results. The best performing algorithm was the MNB, and therefore, it was chosen for the system. The "`util/util_classifier.py`" implements the model training and prediction, and to decrease response time from model training, "`JobLib`" is used to store and retrieve the model, once it has been trained, so that users are not waiting for the classifier to be re-trained after every input, or every system restart.

Initially, a dataset was specifically generated for the classifier. However, even with data bootstrapping, and extra datasets found online, the classifier was not able to properly predict any user responses. However, the classifier was experimentally trained against other datasets, and it was found that using the IMBD review dataset produced a great classifier for the task. Thus, the IMBD review dataset was used as the dataset to train sentiment analysis for the chatbot system.

3 Conversational Design

The conversation design describes the principles which promote user experiences. It is a key aspect of developing conversational systems, as the design aims to create more intuitive and engaging interactions between the system and the users.

3.1 Prompt Design

To design good prompts for users, the prompts must be clear and easy to understand. This can be done through creating more concise prompts, and the usages of conversational markers.

3.1.1 Concise and Clarity

One of the largest outputs the system has is the discoverability task output, where the chatbot regurgitates every task it can do for the user. To counter this the system uses a dictionary, traversing through each task, outputting them into small digestible bullet points. Furthermore, during reservation booking and viewing, the system outputs are differentiated by using different output markers, in order to separate input confirmation data using the ">>" marker, and the next prompt from the "`Hadley:`" marker. This is illustrated in Fig. 4.

3.1.2 Timelines

```
Hadley: I predict that you are trying to make a transportation reservation. Is this correct? (y/n)
User: y

>> Start Transportation Reservation
   Please only type the prompted response, and nothing more!
   Type "cancel" to cancel this reservation at any time.

Hadley: Firstly, what is the name for the reservation?
User: Mint

>> Reservation information added: name - Mint

Hadley: What is the transport for the reservation?
User:
```

Fig. 4. Example of reservation booking prompting, using different markers to show the difference between prompt and other reservation information.

A timeline marker is a conversational marker used to show how far along the user is until a task is completed. This was used in the system's reservation booking task, as the booking requires up to five steps until completion. Fig 5 showed how the system implemented the markers "firstly", "halfway", and "lastly" to help users gauge the amount of work left before a reservation is completed.

```python
    # get (if any) conversational markers, and whether the marker changes the sentence
    4 usages (3 dynamic)
    def get_marker(self):
        # conversational marker: first step
        steps = self.get_steps()
        if steps == 0:
            return "Firstly, ", True
        # conversational marker: halfway mark
        if self.steps_to_complete % 2 == 0:
            if steps == (self.steps_to_complete // 2):
                return "You're halfway done with the reservation! ", False
        else:
            if steps == (self.steps_to_complete // 2) + 1:
                return "Over halfway done! ", False
        # conversational marker: last step
        if steps == (self.steps_to_complete - 1):
            return "And lastly, ", True
        return "", False
```

Fig. 5. Implementation of the timeline marker for the reservation task.

3.1.3 Acknowledgements

An acknowledgement marker is a conversational marker used to show recognition and awareness of user input. The system does this in many places, including acknowledgement of cancellation or confirmation when making a reservation, acknowledgement of small talk replies, and acknowledgement of greetings. Fig. 6 and Fig. 7 portray examples of this.

```
User: exit
Hadley: I predict that you want to exit my program. Is that true? (y/n)
User: n
Hadley: Let's not do that then!
        Reminder that you can say 'goodbye' directly to exit my program!
User:
```

Fig. 6. Example output of the acknowledgement marker when an action is cancelled.

```
User: how are you
Hadley: I'm doing good. Thanks for asking!
        How about yourself?
User: I'm awful
Hadley: I'm sorry if you're not feeling too awesome right now :( Maybe booking a trip to somewhere fun can help?
User:
```

Fig. 7. Example output of the acknowledgement marker when negative sentiment is recognised.

3.1.4 Positive feedback

The system provides positive feedback markers during interactions, such as when the user completes a reservation, when users thank the chatbot, and when the user detects positive sentiment in responses. Fig. 8 shows an example of this.

```
User: how are you
Hadley: I'm doing great, thank you!
User: how are you doing?
Hadley: I'm doing great, thank you!
        How about yourself?
User: I'm great
Hadley: I'm glad to hear you are good!
User:
```

Fig. 8. Example output of the positive response from the chatbot.

3.1.5 Greetings

The chatbot system always greets the users when the application first starts, then again every time the user greets the chatbot. The system also says farewell to the users every time they choose to exit the application. Fig. 9 and Fig. 10 portrays examples of the greetings and farewell the system outputs to the users.

```
Mint: hi
Hadley: Howdy, Mint.
Mint: hello
Hadley: Hello, Mint.
Mint: hi
Hadley: Hey, Mint.
Mint: howdy
Hadley: Hawai'i, Mint.
Mint: hey
Hadley: Hawai'i, Mint.
Mint: hello
Hadley: Hullo, Mint.
Mint:
```

Fig. 9. Example output of the greeting marker greeting the user back.

```
User: goodbye
Hadley: Good day! See you again later!
Hadley: Thank you for keeping me company. I hope you found me useful :)
        See you later!
```

Fig. 10. Example output of the greeting marker to return a farewell to user.

3.2 Discoverability

Discoverability in design principle is concerned with the ease of finding features, information, or functionalities of a system. As the chatbot does not have an easily explorable visual interface, it is very important to apply discoverability, in order to reduce user frustrations and improve quality of user engagement. This system uses discoverability in two ways: main strategy and response strategy.

3.2.1 Main Strategy

The main strategy includes allowing users to type a universal command "help", in order to get a regurgitation of all the possible tasks the system is able to do. From Fig. 11, the figure illustrates how the system can also match other forms of inputs to discoverability as well, allowing more than one way for users to access this discoverability strategy. The figure also shows how the discoverability strategy listed tasks in bullet point instead of paragraph for easier readability.

```
Mint: What can you do?
Hadley: My main task is to assist you in making transportation bookings. (type "make / view reservation")
        However, I can also assist you in other simple chatbot tasks, such as...
        • Intent matching - I can identify and match similarity-based intent matching.
        • Identity management - I can keep track of who you are!
        • Question answering - I can answer factual questions you have using information from my dataset.
        • Small talk - I cam reply to simple greetings and answer questions about the weather!
Mint:
```

Fig. 11. The complete output given by the system when asked to provide it's lists of functionalities.

3.2.2 *Response Strategy*

The response strategy is triggered when an issue occurs. This happens when discoverability is not called, but the user input cannot be accepted. An example of this is when the system unpromptedly gives the user randomised hints of tasks it can perform, when a command that cannot be matched with any system intent is inputted, as shown in Fig. 12. This strategy is also used during reservation making, when the list of available transportation types are explicitly given to the user every time an unknown transportation type has been entered.

```
Mint: I want to bkkking a resort?!
Hadley: I'm sorry, I don't quite understand what you mean.
        Something I can do: assist you in making transportation bookings. (type "make / view reservation", as it is my main task.
Mint: |
```

Fig. 12. The complete output given by the system when asked to provide it's lists of functionalities.

3.3 Error Handling

The system implemented multiple techniques for handling errors, including reprompting, *not* reprompting (ignoring), tiered confidence level confirmations, and dynamic input handling.

3.3.1 *Reprompting vs Ignoring*

The system is designed to reprompt for necessary information. This includes information such as the user's name, and reservation information. The reprompts will provide more information on what input is expected, such as the formatting of the input, allowing users to understand their mistakes and resolve their errors.

However, outside the booking system loop, many errors are not important and can be handled by ignoring them. An example is ignoring empty inputs by waiting to collect the next input instead (see Appendix B.1.1). This is since entering empty inputs is a self-explanatory error that users will intuitively know how to fix, whereas reprompting can be less effective and clutter the text interface.

3.3.2 *Tiered Confidence Level System*

The tiered confidence level system is used in small talk and information retrieval to account for handling possible errors with lower similarity matching. This allows the chatbot to answer differently, depending on how high the input intent matching score (confidence) is. An input with high confidence will automatically be treated as correct, while a slightly lower confidence score will lead to requiring a confirmation. An input with low confidence are rejected, as it is treated as an input error. The implementation of this is shown in section 3.5.

3.3.3 *Dynamic Input Handling*

The chatbot system can handle inputs differently, depending on the type of input it is given. This is done after an input is matched to a certain intent, where they are parsed to check if an input holds more information than just an instruction. These include information for instructions like name changing, reservation making, and reservation viewing. Fig. 13 demonstrates how a more experienced user can make a reservation with all the necessary information in a single input, as opposed to five different prompted inputs. Dynamic handling of data not only allows experienced users to reduce the number of interactions required for an action, but can also ensure important data isn't lost when instructing the chatbot.

3.4 Personalisation

The main personalisation feature in the chatbot system is identity management, as described in section 2.3, where the user's name is stored in the system until the system stops running. Keeping a hold of the user's name can then further improve user experience by simplifying the reservation making/viewing tasks, as they won't have

```
Hadley: Okay! Your name is now Sarah.
Sarah: I want to make a reservation for an XL taxi for 12/1/2025 at 6:30 using email sarahCapenter23@gmail.com

>> Start Transportation Reservation
   Please only type the prompted response, and nothing more!
   Type "cancel" to cancel this reservation at any time.

>> Name detected:
>> Reservation information added: name - Sarah

>> Transportation detected:
>> Reservation information added: transport - xl taxi

>> Transportation detected:
>> Reservation information added: transport - taxi

>> Date detected:
>> Reservation information added: date - 12/1/2025

>> Time detected:
>> Reservation information added: time - 6:30

>> Email detected:
>> Reservation information added: email - sarahCapenter23@gmail.com
Hadley: All information entered successfully!

        name: Sarah
        transport: taxi
        date: 12/1/2025
        time: 06:30
        email: sarahCapenter23@gmail.com
```

Fig. 13. An example of how a user can input more information into the command to simplify the reservation process.

to keep reentering their names. The chatbot also changes the text interface to reflect the user's name, as seen in Fig. 14, adding a visual element to the personalisation.

```
Hadley: Good morning! My name is Hadley, your friendly chatbot. I am mainly here to assist you in making transportation bookings.
        However, I can also assist you in other simple chatbot tasks :)
        Note - when you are done with me, just say "goodbye" to let me know!
User: Hello
Hadley: Hullo! What is your name?
User: Sarah
Hadley: Nice to meet you Sarah. I'm Hadley - but you already know that :)
Sarah: How are you?
Hadley: I'm not too shabby!
        How about yourself?
Sarah: |
```

Fig. 14. An example of how the text interface is changed once a user's name is given to create personalisation.

Another personalisation design is that when the system first starts running, it will greet users differently, depending on the local time, i.e. the chatbot will greet the user with "Good morning" before 1pm, "Good afternoon" between 1pm and 6pm, and "Good evening" after 6pm. Similarly, the small talk's weather retrieval feature is personalised to the local weather in Nottingham, and only gives the weather data of the current day.

3.5 Confirmation

The system uses different types of confirmations, including generic confirmation, implicit confirmation, explicit confirmation, and tiered confidence level confirmation (as mentioned in section 3.3.2).

3.5.1 Generic Confirmation

This confirmation method is used when trying to cancel the reservation that is currently being booked, and is illustrated in Fig. 15

```
Hadley: Firstly, what is the name for the reservation?
User: cancel
Hadley: Are you sure you want to cancel the reservation? This action cannot be undone. (y/n)
User: n
Hadley: Let's not do that then!
```

Fig. 15. An example of generic confirmation.

3.5.2 Implicit Confirmation

This confirmation method is used when the user's input confidence level is high, but misidentification is still possible. An example of this is shown in Fig. 16.

```
User: how much is tablespoon?
Hadley: To answer "how much is 1 tablespoon of water", This tablespoon has a capacity of about 15 mL.
User:
```

Fig. 16. An example of implicit confirmation.

3.5.3 Explicit Confirmation

This confirmation method is used when the user's input confidence level is not very high, but is still identified as something. An example of this is shown in Fig. 17.

```
User: What will?
Hadley: I found the question "what is in a will" in my database. Is that what you mean? (y/n)
User:
```

Fig. 17. An example of explicit confirmation.

3.5.4 Tiered Confidence Level Confirmation

This confirmation method uses different confidence levels to ensure the system responds appropriately, with different methods of confirmation (implicit and explicit). This is implemented in two tasks: small talk and information retrieval. Fig. 18 shows an implementation of this.

```
# REPLY: retrieve the correct answer for a question
1 usage
def reply(content, id, confidence):
    # return answer confidently
    if (confidence >= 0.90):
        return content["Answer"].iloc[id]
    # return answer with implicit confirmation
    if (confidence >= 0.80):
        return (f"To answer \"{content["Question"].iloc[id]}\", {content["Answer"].iloc[id]}")
    # return both question and answer, to be explicitly asked to user when confidence level between 70-80%
    return (content["Question"].iloc[id], content["Answer"].iloc[id])
```

Fig. 18. An example of tiered confidence level confirmation implemented in small information retrieval.

3.6 Content Tracking

The main function of content tracking is tracking information for reservation making. This is done through implementing a "`Reservation`" class, which creates a dictionary of data when the class is instantiated as an object. This implementation is shown in Fig. 19, where other variables in the class such as “`completed`” and “`steps_to_complete`” can also be seen, and are also used for context tracking.

```
# class to manage a reservation
2 usages
class Reservation:
    # initialise all booking information as None
    def __init__(self, transport_types):
        # set up data for reservation
        self.transport_types = transport_types
        self.data = {
            'name': "",
            'transport': "",
            'date': "",
            'time': "",
            'email': ""
        }
        # set reservation tracker
        self.completed = False
        self.steps_to_complete = len(self.data)
```

Fig. 19. An implementation of context tracking through using object-oriented programming and dictionary.

4 Evaluation

To ensure the chatbot system works as intended, it must be thoroughly tested. Therefore, two types of testing were used to evaluate this system: usability testing and performance testing.

4.1 Usability Testing

This test focuses on user experiences, and is done by giving participants a set of instructions that they must complete. The participants are timed, and must rate their experiences using the Chatbot Usability Questionnaire (CUQ) [2] afterwards. A total of 5 participants took part in this test.

4.1.1 Test Instruction

The following instructions are given to the participants. They are aimed to be completed within 5 minutes:

(1) Add a name to the chatbot system.
(2) Change your name to the ”all-cap” version of the name.
(3) Ask the chatbot a question from the QA_dataset.csv given, and check that the answer matched the question.
(4) Make a transportation reservation.
(5) View the reservation that you just made.
(6) Ask about the weather.
(7) Exit the program.

4.1.2 Participant Timing Results

Fig. 20 illustrates the time taken for participants to complete the given instructions.

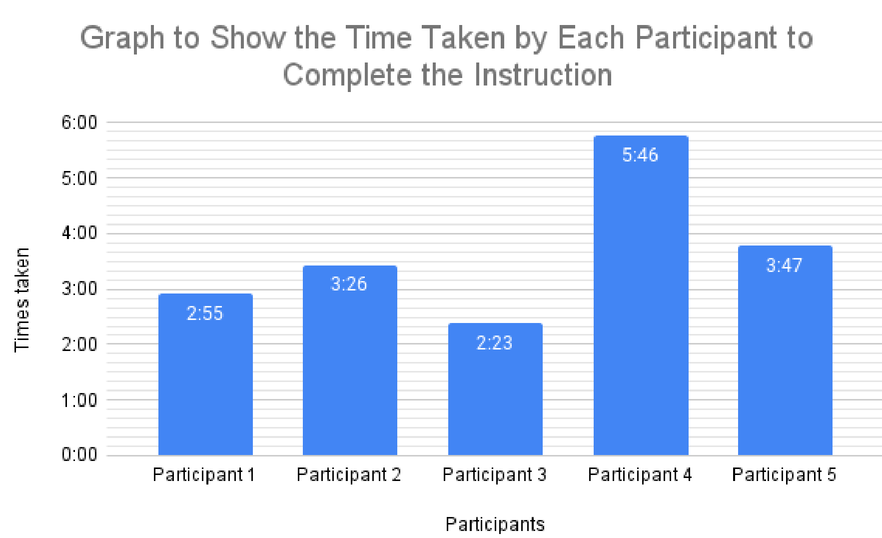

Fig. 20. The Graph of Times Taken for Each Participant to Complete the Instructions Given.

4.1.3 Chatbot Usability Questionnaire Results

Fig. 21 shows the scoring from each participant, and their final calculated percentages, and Fig. 22 is a stacked bar chart visualisation of the results.

Questions	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5
Question 1	5	5	5	5	5
Question 2	1	1	1	1	1
Question 3	5	5	5	5	5
Question 4	1	1	1	1	1
Question 5	5	4	4	3	5
Question 6	1	1	1	3	1
Question 7	5	5	5	4	4
Question 8	1	1	1	1	1
Question 9	5	4	5	3	4
Question 10	1	1	1	2	2
Question 11	5	5	5	4	5
Question 12	1	1	1	1	1
Question 13	5	5	5	4	5
Question 14	1	1	1	1	1
Question 15	5	5	5	4	5
Question 16	1	1	1	2	1
QUC Scores	100.00%	96.88%	98.44%	81.25%	95.31%

Fig. 21. Chatbot Usability Questionnaire Scores from Participants.

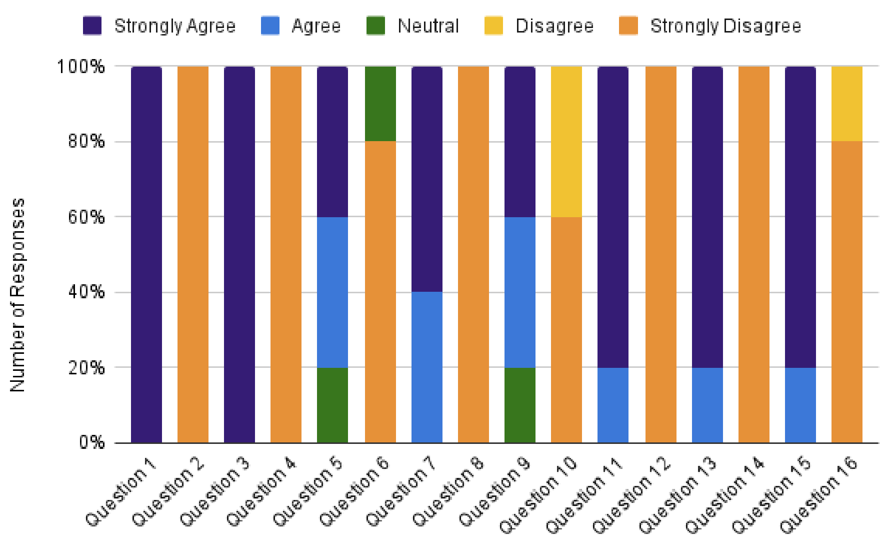

Fig. 22. Stacked Bar Chart of the Chatbot Usability Questionnaire Results.

4.2 Performance Testing

This test focuses on the objective evaluation of the chatbot system, and will involve testing the sentiment analysis classifier, using matrices including confusion metrix, precision, recall, and F1-score

4.2.1 Evaluation with Train-Test set

Using the IMBD dataset, which was originally used to train the system's sentiment analysis, the data is split into train and test sets, and was evaluated. Fig. 23 shows the results of this testing.

```
Confusion Matrix:
 [[213   6]
 [ 15 216]]
Classification Report:
              precision    recall  f1-score   support

    negative       0.93      0.97      0.95       219
    positive       0.97      0.94      0.95       231

    accuracy                           0.95       450
   macro avg       0.95      0.95      0.95       450
weighted avg       0.95      0.95      0.95       450
```

Fig. 23. Performance evaluation of the sentiment analysis using the test-train set with the IMBD dataset.

4.2.2 Evaluation with new unseen dataset

However, to ensure the classifier can also perform well with its intended task, a new dataset is generated (see Appendix C.1.1), to evaluated its generalisation with small talk sentiment analysis. Using this newly generated dataset for small talk responses, the classifier's prediction is evaluated. Fig. 24 shows the results of this testing.

```
Confusion Matrix:
 [[21  4]
 [ 5 20]]
Classification Report:
              precision    recall  f1-score   support

    negative       0.81      0.84      0.82        25
    positive       0.83      0.80      0.82        25

    accuracy                           0.82        50
   macro avg       0.82      0.82      0.82        50
weighted avg       0.82      0.82      0.82        50
```

Fig. 24. Performance evaluation of the sentiment analysis against a newly generated dataset for small talk responses.

5 Discussion

5.1 Reflection on Evaluation Results

5.1.1 Usability Testing

Through evaluating participates' timed performances and their CUQ scores, insights into user experiences were gathered.

By timing participants to follow a set of instructions, the chatbot's ease of use and understandability were assessed. From the result, four out of five participants were able to complete the instructions within the expected time, with the overall average time taken of around 3 minutes 39 seconds, and the standard deviation of around 1 minute 18 seconds. This provides evidence of ease to use, as majority of participants completed the instructions with more than a minute of time to spare, all without prior preparation. This included easily completing larger tasks such as making transportation reservations, which was initially thought would be more difficult for most users.

However, one participant struggled greatly while making their reservation. This was due to the initial insufficient data on the formatting in the prompt, leading them to be unsure how to enter their data. Although originally designed to be short and concise, the reservation prompts did not have enough data for this participant, causing a lot more time to be spent on it.

The qualitative feedback collected from the CUQ also showed strength in the systems' usability, providing an average score of over 94%. These highlighted strengths in conversational aspects such as the friendly and non-robotic personality of the chatbot, along with its abilities to handle errors, and guidance to subdue confusion.

On the other hand, the CUQ also provided more troublesome issues with the system. Looking at Fig. 22, questions with less ideal scores were pointed out to Question 5 and Question 9 having the lowest scores. These questions referred to the chatbot's explanations and understanding of user inputs. This result provided similar themes of prompt design to the previous issue mentioned. Furthermore, this score alluded to the chatbot's more rigid acceptance of input formats, which also caused frustration to some participants.

From these testing results, flaws from certain aspects of the chatbot's system were revealed. Looking forward, the system's prompt design should be addressed and redefined, in order to balance between the benefit of conciseness, and understandability. Furthermore, more development of input acceptance should be explored. A formatting frustration that was voiced by a participant was the chatbot's inability to accept dates with two digits year, instead of the full four digit. By exploring strategies which acknowledges more types of formatting, this can also be another solution to the current prompts being short and concise.

5.1.2 Performance Testing

To evaluate the chatbot's sentiment analysis, f1-score will be used, as this evaluation metric takes into account both the precision and recall of the classifier.

As the Multinomial Naive Bayes classifier has been previously tested against other classifier algorithms (see section 2.7.2), the F1-score was expected to be over 80%. However, through data bootstrapping, the performance testing achieved the F1-score of 95%, a 10% increase from the initial experiment. This shows a strong reflection on the classifier's sentiment analysis abilities.

However, evaluating the small talk against the new dataset revealed a decrease in classification performances, with an overall f1-score of 82%. This result portrays the possible incompatibility of the training dataset for the classifying job.

From these results, future improvements of the chatbot system, can be to gather better datasets that are more specific to the types of application that the classifier is using. Furthermore, deep learning algortihms such as Random Forest and Multi-layer Perceptron could be explored as alternative algorithms, to experiment against the current algorithm, to possibly find a model that may yield better results. These improvement will ensure the chatbot's sentiment analysis can will be able to better classify user inputs in the future, and supporting higher quality user interactions.

5.2 Reflection on the Project Overall

The development of the artificial intelligence system was aimed at creating a versatile and user-friendly chatbot that is capable of handling diverse tasks, including intent matching, identify management, transactional booking system, information retrieval, and small talk.

Through this project, better comprehension of natural language processing and design principles were achieved. Understanding of the chatbot system, and its integration of database and machine learning were also consummated. This is demonstrated in the chatbot's robust performances in areas such as intent matching, transactional reservation tasks, and sentiment analysis.

The evaluation provided important insights into the chatbot's performances, including strengths such as its friendliness and relevancy of responses to users. However, weaknesses were identified in areas including prompt design and error handling of unformatted inputs. However, further improvements suggested in the discussion section will be able to reduce these issues.

Overall, the project was successful in creating a cohesive chatbot system with the ability to perform all required tasks, and follows proper design principles, leading to a satisfactory solution for the project,

References

[1] Madhur Jain, Shilpi Jain, Shruti Daga, and Roshni. 2024. Leveraging AI in Predictive Analytics. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 10, 3 (2024), 50–60. https://ijsrcseit.com/index.php/home/article/view/CSEIT2410314/CSEIT2410314 Accessed: 2024-12-8.

[2] Ulster University. 2024. CUQ: Comprehensive Usability Questionnaire. https://www.ulster.ac.uk/research/topic/computer-science/artificial-intelligence/projects/cuq Accessed: 2024-12-10.

[3] Wikipedia contributors. 2024. Cosine Similarity — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Cosine_similarity Accessed: 2024-12-8.

A Chatbot Architecture

A.1 Intent Matching

A.1.1 Cosine Similarity Matching

```
# match input with matrix instance using cosine similarity
4 usages
def process_query_with_matrix(query, file_name, extension, threshold):
    # get matrix and vectoriser
    content, QA_matrix, tfidf_vectoriser = process_question_dataset(file_name, extension)

    # preprocess query
    query_preprocessed = preprocess_text(query)

    # get cosine similarities
    query_tfidf = tfidf_vectoriser.transform([query_preprocessed]).toarray()[0]
    cos_sim = []
    for i in range(len(QA_matrix)):
        denom = norm(query_tfidf) * norm(QA_matrix.iloc[i])
        if denom != 0:
            cos_sim.append(dot(query_tfidf, QA_matrix.iloc[i]) / denom)
        else:
            cos_sim.append(0)

    # get max cosine
    max_sim_value = max(cos_sim)
    # get randomised
    if (max_sim_value > threshold):
        max_sim_indices = [i for i, value in enumerate(cos_sim) if value == max_sim_value]

        id = max_sim_indices[random.randint( a: 0, len(max_sim_indices) - 1)]

        return id, content, max_sim_value
    return None, content, max_sim_value
```

Fig. 25. The code snippet of cosine similarity function used in intent matching.

A.1.2 Simple Token Similarity Matching

```
# match input with score compared to given list
2 usages
def process_query_with_list(user_input, list, threshold):
    score = 0
    input_tokens = preprocess_small(user_input)

    if(len(input_tokens) == 0):
        return False

    for token in input_tokens:
        if token in list:
            score += 1
    score /= len(input_tokens)
    return True if score > threshold else False
```

Fig. 26. The code snippet of simple binary similarity function used in intent matching.

A.2 Transaction

```
User: view reservation

>> Start Reservation Viewing
   Please only type the prompted response, and nothing more!
   Type "cancel" to cancel this reservation at any time.

Hadley: Can I please get your name for this reservation?
User: Gabi
Hadley: Can I please get your reservation ID?
Gabi: 0003
Hadley: I found your reservation information!

>> Reservation information:
>> id:  003
>> name: Gabi
>> transport: coach
>> date: 28/12/2024
>> time: 08:25
>> email: Gab123@gmail.com
```

Fig. 27. The full interaction of reservation viewing transaction, without prior input data.

A.2.1 Reservation

```
Hadley: Good morning! My name is Hadley, your friendly chatbot. I am mainly here to assist you in making transportation bookings.
        However, I can also assist you in other simple chatbot tasks :)
        Note - when you are done with me, just say "goodbye" to let me know!
User: make reservation

>> Start Transportation Reservation
   Please only type the prompted response, and nothing more!
   Type "cancel" to cancel this reservation at any time.

Hadley: Firstly, what is the name for the reservation?
User: Gabi

>> Reservation information added: name - Gabi

Hadley: What is the transport for the reservation?
User: coach

>> Reservation information added: transport - coach

Hadley: What is the date for the reservation?
User: 28/12/2024

>> Reservation information added: date - 28/12/2024

Hadley: Over halfway done! What is the time for the reservation?
User: 8:25

>> Reservation information added: time - 08:25

Hadley: And lastly, what is the email for the reservation?
User: Gab123@gmail.com

>> Reservation information added: email - Gab123@gmail.com
Hadley: All information entered successfully!

        name: Gabi
        transport: coach
        date: 28/12/2024
        time: 08:25
        email: Gab123@gmail.com

Hadley: Please confirm that this information is correct. (y/n)
        Note that the reservation will have discarded if confirmation not approved
User: y

>> Reservation saving in progress, please do not close the program...
>> Reservation saving completed.

Hadley: Your reservation for coach is successfully recorded!
        Your reservation ID is 0003. Please note this down, as it will be needed to access your reservation later!
```

Fig. 28. The full interaction of reservation making transaction, without prior input data.

A.2.2 Reservation

A.2.3 Reservation

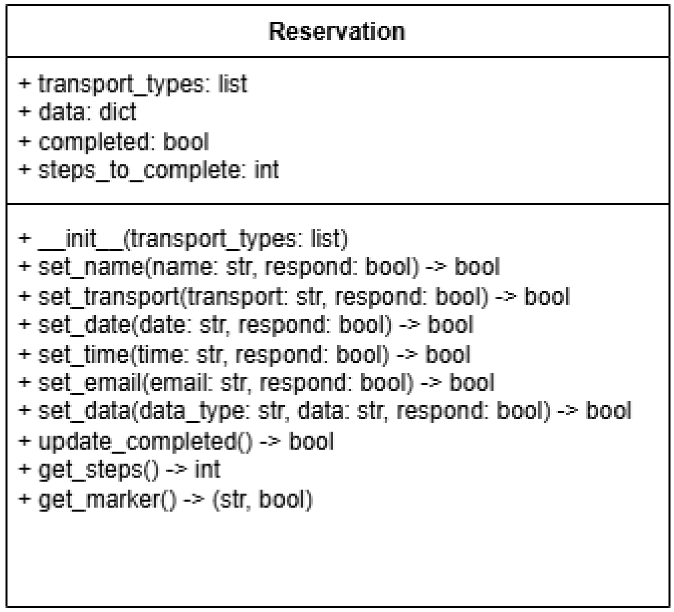

Fig. 29. The UML class diagram of the Reservation class used to create reservations in booking.py.

A.3 Sentiment Analysis

A.3.1 Model Algorithm Experiment

```
def loadData():
    import os
    label_dir = {
        "positive": "data/positive",
        "negative": "data/negative"
    }

    data = []
    labels = []

    for label in label_dir.keys():
        for file in os.listdir(label_dir[label]):
            filepath = label_dir[label] + os.sep + file
            with open(filepath, encoding='utf8', errors='ignore', mode='r') as review:
                content = review.read()
                data.append(content)
                labels.append(label)

    return data, labels

def stemmed_words(doc):
    from nltk.stem.snowball import PorterStemmer
    from sklearn.feature_extraction.text import CountVectorizer
    from nltk.corpus import stopwords

    p_stemmer = PorterStemmer()
    analyzer = CountVectorizer(stop_words=stopwords.words('english'),
                ngram_range=(1, 2)).build_analyzer()
    return (p_stemmer.stem(word.lower()) for word in analyzer(doc))

def evalMetrics(count_vect, tfidf_transformer, name, classifier, X_test, y_test):
    from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

    X_new_counts = count_vect.transform(X_test)
    X_new_tfidf = tfidf_transformer.transform(X_new_counts)

    y_pred = classifier.predict(X_new_tfidf)

    # get metric scores
    cm = confusion_matrix(y_test, y_pred)
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, pos_label='positive')
    print(f"Evaluation completed for {name}")
```

```
    return [name, cm, accuracy, f1]

def compareAlgor():
    from sklearn.model_selection import train_test_split
    from sklearn.feature_extraction.text import CountVectorizer
    from sklearn.feature_extraction.text import TfidfTransformer
    from sklearn.linear_model import LogisticRegression
    from sklearn.naive_bayes import MultinomialNB
    from sklearn.svm import SVC
    from sklearn.tree import DecisionTreeClassifier

    data, labels = loadData()
    X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.25,
                                        stratify=labels, random_state=1)

    # create bag-of-words representation
    count_vect = CountVectorizer(analyzer=stemmed_words)
    X_train_counts = count_vect.fit_transform(X_train)
    tfidf_transformer = TfidfTransformer(use_idf=True,
                        sublinear_tf=True).fit(X_train_counts)
    X_train_tf = tfidf_transformer.transform(X_train_counts)

    # create classifier
    lgr = LogisticRegression(random_state=0).fit(X_train_tf, y_train)
    mnb = MultinomialNB().fit(X_train_tf, y_train)
    svc = SVC(random_state=0).fit(X_train_tf, y_train)
    dtc = DecisionTreeClassifier(random_state=0).fit(X_train_tf, y_train)

    # evaluate
    eval = []
    eval.append(evalMetrics(count_vect, tfidf_transformer,
                            "LogisticRegression", lgr, X_test, y_test))
    eval.append(evalMetrics(count_vect, tfidf_transformer,
                            "MultinomialNB", mnb, X_test, y_test))
    eval.append(evalMetrics(count_vect, tfidf_transformer,
                            "SVC", svc, X_test, y_test))
    eval.append(evalMetrics(count_vect, tfidf_transformer,
                            "DecisionTreeClassifier", dtc, X_test, y_test))

    # print values
    eval.sort(key=lambda x: x[2]+x[3], reverse=True)
    for algor in eval:
        print(f"{algor[0]}:")
        print(f" Accuracy:\t{algor[2]}")
        print(f" F1-Score:\t{algor[3]}")
        print(f" Confusion Matrix:\n {algor[1]}")
```

A.3.2 Model Algorithm Experiment Result

```
Evaluation completed for LogisticRegression
Evaluation completed for MultinomialNB
Evaluation completed for SVC
Evaluation completed for DecisionTreeClassifier
MultinomialNB:
 Accuracy:  0.856
 F1-Score:  0.85
 Confusion Matrix:
 [[112  13]
 [ 23 102]]
LogisticRegression:
 Accuracy:  0.848
 F1-Score:  0.8455284552845529
 Confusion Matrix:
 [[108  17]
 [ 21 104]]
SVC:
 Accuracy:  0.844
 F1-Score:  0.8408163265306122
 Confusion Matrix:
 [[108  17]
 [ 22 103]]
DecisionTreeClassifier:
 Accuracy:  0.592
 F1-Score:  0.5641025641025641
 Confusion Matrix:
 [[82 43]
 [59 66]]

Process finished with exit code 0
```

Fig. 30. The resulting output from running the the model algorithm experiment.

B Conversational Design

B.1 Error Handling

B.1.1 Error Handling by Ignoring

```
Gabi: how water tablespoon
Hadley: I found the question "how much is 1 tablespoon of water" in my database. Is that what you mean? (y/n)
Gabi:
Gabi:
Gabi:
Gabi:
Gabi: no
Hadley: Let's not do that then!
Gabi:
```

Fig. 31. An example of error handling by ignoring a blank input from user.

C Evaluation

C.1 Performance Testing

C.1.1 Small talk Response

```
new_data_positive = ["I'm great", "I feel happy", "Everything is awesome", "I'm doing well today", "I'm so
 happy today", "I'm feeling fantastic", "I'm in a good mood", "I'm feeling positive", "I'm in high spirits",
 "I'm feeling wonderful", "Today is a good day", "I'm in a very good mood", "I'm feeling absolutely great",
 "I'm feeling excellent", "I'm doing superb", "Feeling really positive today", "I'm doing wonderfully", "I'm
 feeling fabulous", "I'm doing awesome, thanks", "I'm feeling pretty fantastic", "I'm in a great place", "I'm
 loving it", "I'm feeling energised", "I'm having an incredible day", "I'm doing amazing"]

new_data_negative = ["I'm feeling awful", "I'm not great today", "I'm pretty sad", "I'm feeling down", "I feel
 awful", "I'm feeling terrible", "I'm doing poorly", "I’m having a tough day", "I'm feeling depressed", "I'm
 feeling miserable", "Everything is bad", "I'm feeling so negative", "I'm not doing okay today", "I'm really
 upset today", "I'm feeling upset", "Everything is bad", "I'm feeling very down", "Everything is just bad",
 "Today is a bad day", "I'm feeling really negative", "I'm in a bad place", "I am not good at all", "I am so
 bad", "I feel very bad right now", "I am feeling really down"]
```

Fig. 32. The newly generated small talk response for classifier evaluation.