Developing an Intelligent Chatbot: Final Report

Group UG09- Natural Unintelligence:
Steven Diep (100263580)
Sam Humphreys (100251448)
Martin Siddons (100225776)

Introduction

This technical report aims to walk the reader through our process in creating a chat bot designated for booking train tickets, predicting train delays, and providing sufficient customer service making the chat bot easier to use. Our group named Natural Unintelligence consists of three members, Sam Humphreys, Martin Siddons, and Steven Diep. The report covers how we gathered an understanding of how a chat bot should be made and their necessary requirements through background research, and how the work was allocated between three undergraduate students. We also discuss the tools we used to build each of the several parts of the chat bot and its integration, the parts mentioned consists of the reasoning engine (Pyknow), natural language processing (Spacy), prediction model (sklearn), web scraping (beautiful soup), database (SQLite3), user interface (Flask), speech to text (Javascript). Through reading this report you should expect to gain a grasp of how to build a chatbot from the tools we used and understand the project directions we took collectively.

Initial plan

To gain valuable insight on how to complete this assignment efficiently, we organised a weekly meeting in covering each part of the chat bot we needed to implement whilst using a shared document to record for potential future purposes. Once we have figured out each component and their requirements, we then decided who is most suited for completing each part. A private repository on GitHub was also shared between the three of us to have a cohesive workflow while tracking each other's progress.

Distribution of workload

The decision in workload was to distribute each part evenly, this was the decided assignment:

- Sam Humphreys
    - User interface
    - Natural language processing
    - Speech to text
- Steven Diep
    - Reasoning engine
    - Knowledge base
- Martin Siddons
    - Prediction model
    - Database
    - Web scraping

The percentage contribution is agreed to be 33%

Initial design

Following the initial analysis, we discussed what the basic system would look like and what information each component of the system would send and receive. We then mapped this information to a functional diagram, as shown below.

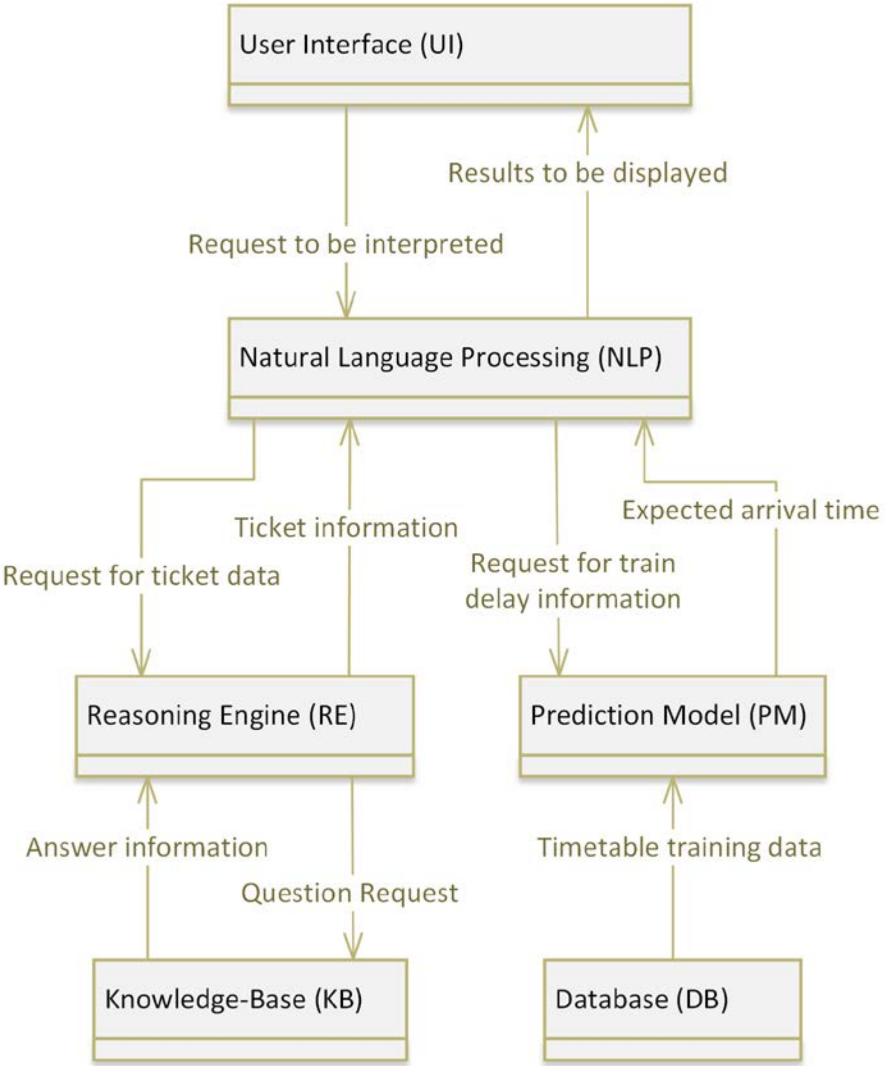


Figure 1: Initial Design Diagram

Related work

This section covers the background work we did on developing a chat bot system, it enables us to have a clearer vision in what we would have to implement in the chat bot and the order in which we should do so.

Background research

To figure out what is necessary to implement for the chat bot we should delve into guidelines that explores the many features a chat bot is required to have. The fundamentals of a chat bot should consist of appropriate feedback to the end user and matching their responses with a decided set of dialogs. An example would be, the user responding "Hello!", the bot should then recognise the user's dialog and respond appropriately through understanding that it is a greeting thus prompting the bot to either respond with "Hey!", "Hello!", or "Hi". A feasible way in achieving responses from complex dialog from the end user is using natural language processing and understanding, it allows the bot to pick up on information that are concatenated within long sentences. The bot must only serve its intended purpose and be transparent to the user what services it provides https://www.microsoft.com/en-us/research/uploads/prod/2018/11/Bot_Guidelines_Nov_2018.pdf.

Chat bots are also used for a multitude of reasons other than the assignment we were set; they serve use to those running businesses with a large customer base. Building a knowledge base suited towards customer queries regarding the business can alleviate the stress put upon a human customer service team, through answering simple queries such as with faster response time can greatly improve on the quality of customer satisfaction https://www.groovehq.com/blog/building-knowledge-base. Chat bots are particularly useful to those who rely on answering the same query in multiple instances such as ordering a specific product or service, the bot can also be adjusted to add more questions and answers in the case of the business expanding.

Aim and Objectives

For our initial analysis, we discussed what features were necessary for the system to work and what extra features we believe users of the system would like to see. We have compiled these features into the MoSCoW Analysis below.

- Must
    - Have a way for the user to communicate with the system.
    - Perform Natural Language Processing to understand user requests.
    - Display appropriate and accurate answer to user, also handle bad questions.
    - Contain one model to predict train delays.
    - Use a Reasoning Engine to find ticket data.
    - Hold ticket data in the Knowledgebase.
    - Limit bot to its purpose only. Do not engage in race, gender, politics, and religion.
- Should
    - A UI which mimics a chat program
    - Utilise multiple models to predict train delays.
    - The ability to understand and correct user misspellings.
    - Offer a detailed explanation of the purpose and operation of the bot.
    - Ability to remember previous search details to help predict future searches, i.e., understanding that the user travels from Norwich so Norwich would be classed as the default “from” station after a couple searches.
- Could

    - Knowledge Acquisition system
    - Text-to-Speech and Speech-to-Text system
    - Clickable links in chatbot replies to change specific details (from / to / date etc.)
    - Clickable links in chatbot text to quickly move into certain queries.
    - Support for understanding other ticket types such as Child or Railcard holder
    - Ability to scrape rail FAQs for Q+A functionality. E.g., answering “How can I make a complaint about this train?
    - Use of Emoticons to make it more lifelike and friendly.
- Won’t
    - Use GPS system to improve customer service.
    - User satisfaction of the chatbot (feedback mechanism)
    - Smooth hand-off process, hand conversation to human moderator
    - Code of conduct on users, prohibit hate speech, threatening others, and bullying.

Difficulties and Risks

Careful planning should be considered for a lengthy project due to unexpected issues that may arise. Acknowledging these risks ahead may help alleviate or mitigate the negative impact it may have on the project. Here are some possible risks we may have to consider:

- Agreed deadline for a component of the bot is incomplete.
- Poor communication
- Catching an illness or some sort of ailment
- Investing time into the wrong task/path
- Hardware problems
- No access to internet
- Assumptions without no consultation
- Misinterpretation of group members
- Misunderstanding of aim

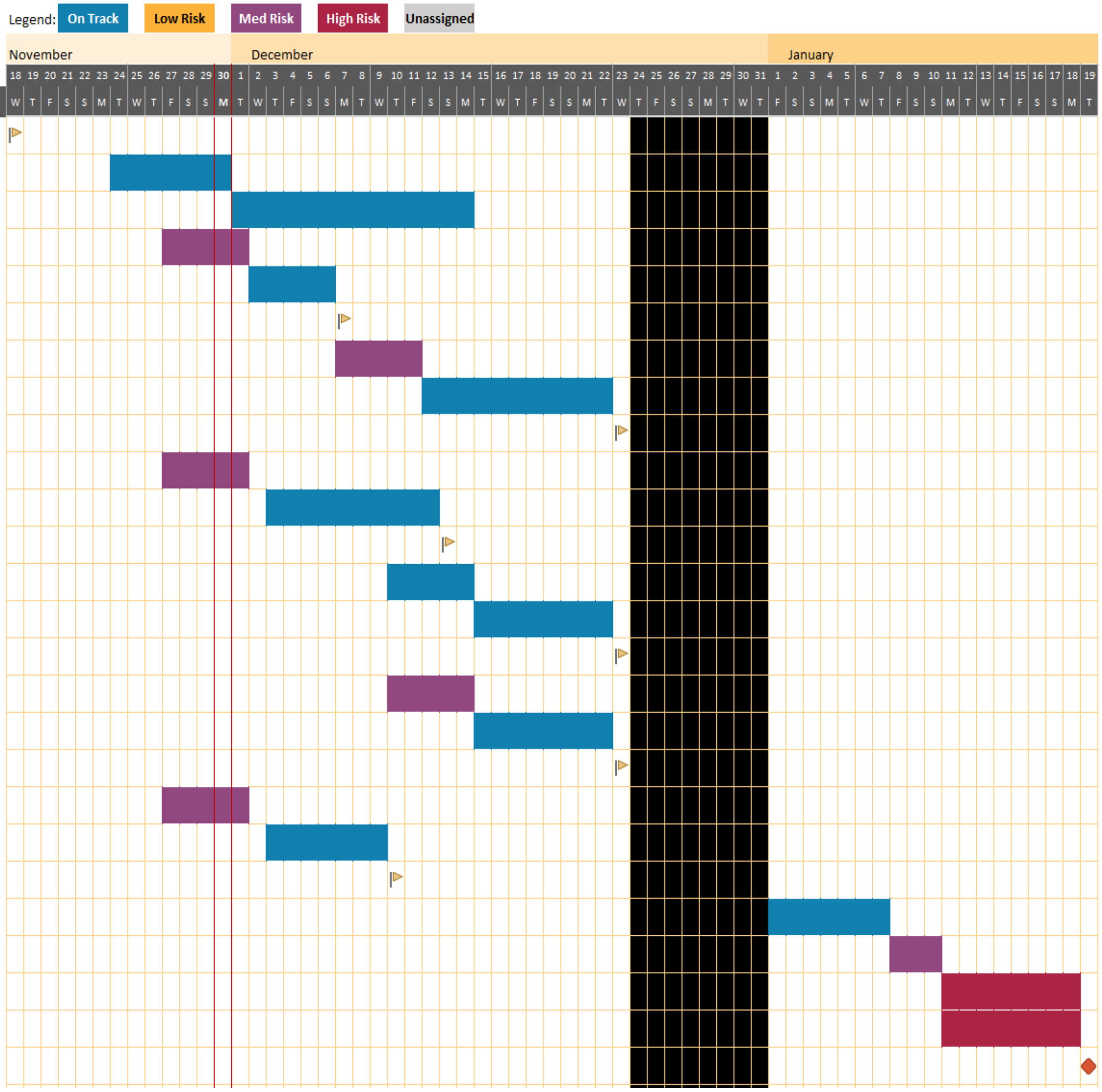
Work Plan

As part of our early work, we devised a Gantt Chart to allow us to plan ahead and understand how much time we could spend on each part of the system and keep us on-track to complete the project by the submission date. We intentionally left some spare time over Christmas to ensure that if we over-ran on our personal sections, that we could make it up during that time.

CW2 - AI ChatBot

Group 9 - Natural Unintelligence

	Milestone Description	Category	Assigned To	Start	No. Days
Project Start	Project started	Milestone	All	18/11/2020	1
	First report	On Track	All	24/11/2020	7
	Second report	On Track	All	01/12/2020	14
User Interface	Initial implementation	Med Risk	Sam	27/11/2020	5
	Additional features & testing	On Track	Sam	02/12/2020	5
	User interface complete	Milestone	Sam	07/12/2020	1
Natural Language Processing	Initial implementation	Med Risk	Sam	07/12/2020	5
	Additional features & testing	On Track	Sam	12/12/2020	11
	NLP complete	Milestone	Sam	23/12/2020	1
Knowledge Base	Initial implementation	Med Risk	Steven	27/11/2020	5
	Additional features & testing	On Track	Steven	03/12/2020	10
	Knowledge base complete	Milestone	Steven	13/12/2020	1
Reasoning Engine	Initial implementation	On Track	Steven	10/12/2020	5
	Additional features & testing	On Track	Steven	15/12/2020	8
	Reasoning engine complete	Milestone	Steven	23/12/2020	1
Prediction Engine	Initial implementation	Med Risk	Martin	10/12/2020	5
	Additional features & testing	On Track	Martin	15/12/2020	8
	Prediction model complete	Milestone	Martin	23/12/2020	1
Database & Web Scraping	Initial implementation	Med Risk	Martin	27/11/2020	5
	Additional features & testing	On Track	Martin	03/12/2020	7
	Database & web scraping complete	Milestone	Martin	10/12/2020	1
Project End	Overflow / extra time	On Track	All	01/01/2021	7
	Integration testing	Med Risk	All	08/01/2021	3
	Prepare final report	High Risk	All	11/01/2021	8
	Prepare presentation	High Risk	All	11/01/2021	8
	Submission	Goal	All	19/01/2021	1



Methods, Tools and Frameworks

Methods

Due to the small project and team size, using Agile would be partially suitable, but having through research we felt that Rapid Application Development (RAD) would be best suited. RAD requires the developer to produce requirements and then move between design and construction of the system to be as flexible as possible to changes we may need to make along the way. It is a methodology that is particularly suitable to small and medium projects that are well-defined and must be developed on a strict timescale. RAD allows us to change course on features and system design should we find a better way of doing things, which is particularly useful as nobody in the group has built an AI project at this scale before, so we anticipate running into many issues bringing the system together.

To stay connected during development, we have a group chat on Microsoft Teams where we can ask questions and get feedback from the rest of the group at any time. We have also scheduled meetings on Tuesdays to inform the group of what we have all done in the previous week, to talk about any findings we have made that may affect integration and to share ideas and get feedback from each other on how to progress in the week following the meeting.

Languages, Packages, Tools

We have chosen Python as our programming language of choice due to there being a large collection of AI libraries available for it, and to allow us to grow our skills in Python.

We will be using SQLite as our database due to it being simple to use, well-supported in Python and fast. Our interface will use Flask as we have some experience using it in the past. For our Knowledgebase and Inference Engine, we are looking to use Pyknow due to its powerful features compared to other KB/IE. Our Prediction Models will be built using scikit-learn due to its ease of use and the many guides available, along with NumPy and Pandas. For NLP, we are looking to use spaCy due to its impressive toolkit, we are also looking to use fuzzywuzzy for fuzzy string matching, pyspellchecker for spellchecking and datefinder for grabbing dates from text. Finally, for web scraping we will be looking to use Scrapy as it appears to be quite feature-rich compared to the alternatives.

Our code will be hosted on a private GitHub repository where we can create Pull Requests to review and test code before it is added to the main project branch. We can also use the Projects feature to track features to be completed and which PRs contained those features. Hosting on GitHub also allows us to use the Issues system to find and track bugs we might find during integration testing.

Development Framework

We spoke about using the Django framework to assist us in producing the UI, but after researching our options we decided that Django would probably be more than we would need, and that our previous experience with Flask made us comfortable that we could produce a working product without a framework. We will look at this in more detail as the project comes together, as it may be that React or NodeJS will be useful to use for building JavaScript elements for the UI.

Design of the Chatbot

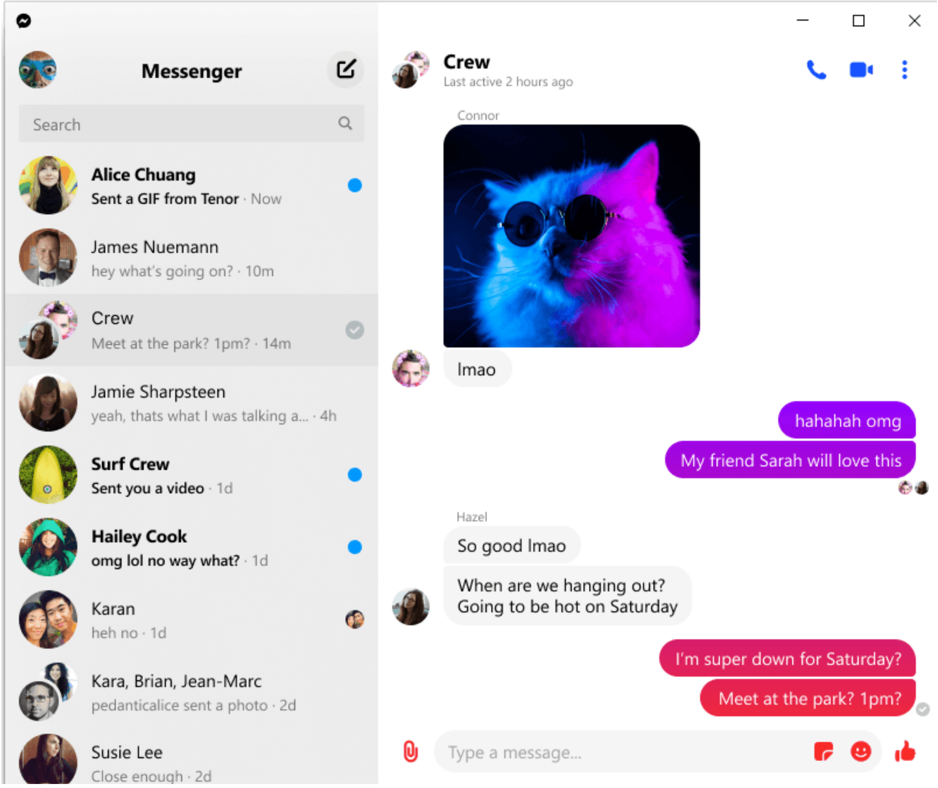
Architecture of the Chatbot

User Interface

Design

As the chatbot is designed for use by a rail company in a customer-facing environment the users will likely interact with it in a web-based manner either through the company website or mobile application. To facilitate this a client-server model is used, with the chatbot itself running in a Python Flask (Lord, 2019) server that clients connect to using a web browser. The client interface is designed to be very familiar for users of chat applications. Looking at existing messaging applications, such as: Microsoft Skype, Facebook Messenger, and Microsoft Teams, they all possess some common features:

- Multiple ways to distinguish messages from different senders (message bubbles, colour, position, profile picture)
- A margin is used for long messages so they do not fill the entire chat window.
- The main controls are located at the bottom of the window alongside the message input area
- Time and date the message was sent
- Images and other files can be sent in-window
- Emojis



Alice Chuang
Sent a GIF from Tenor · Now

James Nuemann
hey what's going on? · 10m

Crew
Meet at the park? 1pm? · 14m

Jamie Sharpsteen
yeah, thats what I was talking a... · 4h

Surf Crew
Sent you a video · 1d

Hailey Cook
omg lol no way what? · 1d

Karan
heh no · 1d

Kara, Brian, Jean-Marc
pedanticalice sent a photo · 2d

Susie Lee
Close enough · 2d

Connor

lmao

hahahah omg

My friend Sarah will love this

Hazel

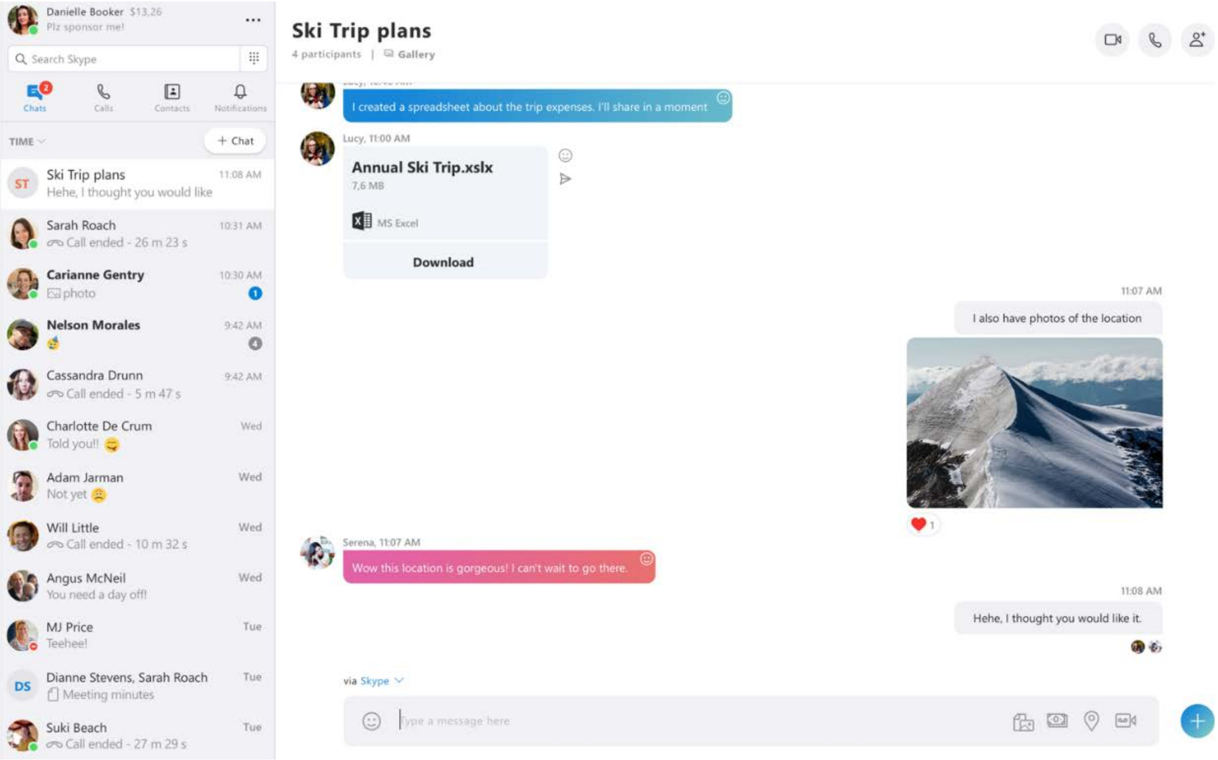
So good lmao

When are we hanging out?
Going to be hot on Saturday

I'm super down for Saturday?

Meet at the park? 1pm?

Figure 2: Facebook Messenger Interface



Ski Trip plans 11:08 AM
Hehe, I thought you would like

Sarah Roach 10:31 AM
Call ended - 26 m 23 s

Carianne Gentry 10:30 AM
photo

Nelson Morales 9:42 AM

Cassandra Drunn 9:42 AM
Call ended - 5 m 47 s

Charlotte De Crum Wed
Told you!!

Adam Jarman Wed
Not yet

Will Little Wed
Call ended - 10 m 32 s

Angus McNeil Wed
You need a day off!

MJ Price Tue
Teehee!

Dianne Stevens, Sarah Roach Tue
Meeting minutes

Suki Beach Tue
Call ended - 27 m 29 s

I created a spreadsheet about the trip expenses. I'll share in a moment

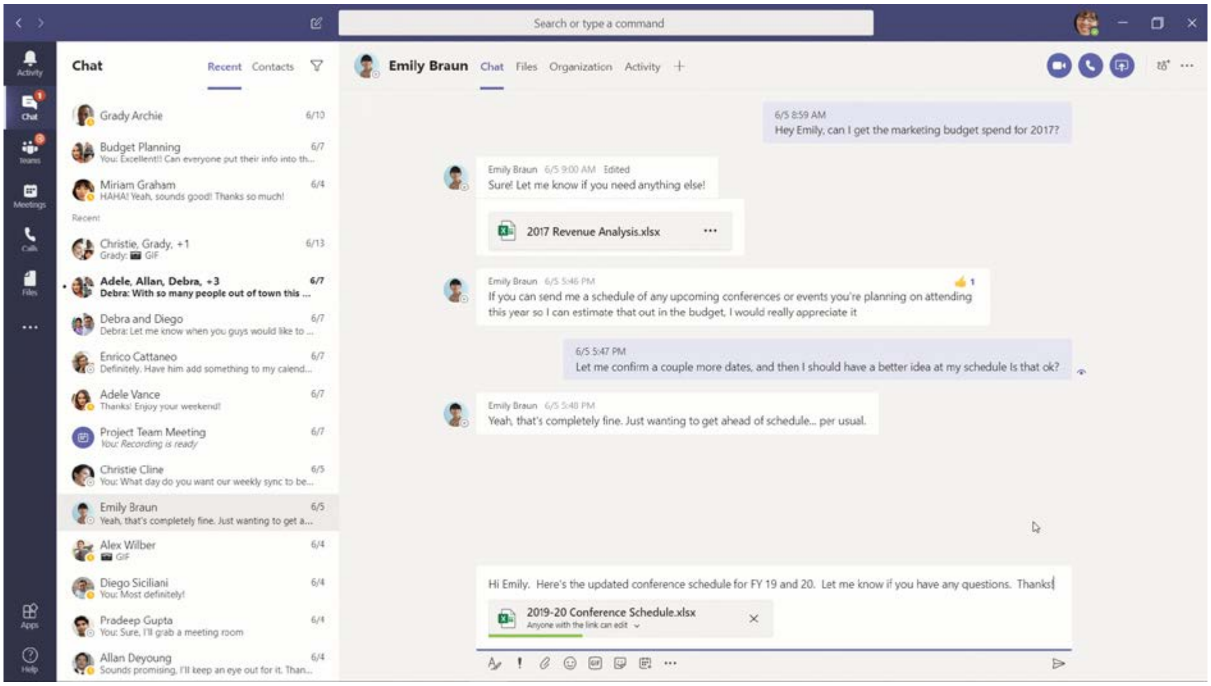
Lucy, 11:00 AM
Annual Ski Trip.xslx
7,6 MB

11:07 AM
I also have photos of the location

Serena, 11:07 AM
Wow this location is gorgeous! I can't wait to go there.

11:08 AM
Hehe, I thought you would like it.

Figure 3: Skype Messenger Interface



Grady Archie 6/10

Budget Planning 6/7
You: Excellent!! Can everyone put their info into th...

Miriam Graham 6/4
HAHA! Yeah, sounds good! Thanks so much!

Recent

Christie, Grady, +1 6/13
Grady: GIF

Adele, Allan, Debra, +3 **6/7**
Debra: With so many people out of town this ...

Debra and Diego 6/7
Debra: Let me know when you guys would like to ...

Enrico Cattaneo 6/7
Definitely. Have him add something to my calend...

Adele Vance 6/7
Thanks! Enjoy your weekend!

Project Team Meeting 6/7
You: Recording is ready

Christie Cline 6/5
You: What day do you want our weekly sync to be...

Emily Braun 6/5
Yeah, that's completely fine. Just wanting to get a...

Alex Wilber 6/4
GIF

Diego Siciliani 6/4
You: Most definitely!

Pradeep Gupta 6/4
You: Sure, I'll grab a meeting room

Allan Deyoung 6/4
Sounds promising, I'll keep an eye out for it. Than...

6/5 8:59 AM
Hey Emily, can I get the marketing budget spend for 2017?

Emily Braun 6/5 9:00 AM Edited
Sure! Let me know if you need anything else!

2017 Revenue Analysis.xlsx

Emily Braun 6/5 5:46 PM
If you can send me a schedule of any upcoming conferences or events you're planning on attending this year so I can estimate that out in the budget. I would really appreciate it

6/5 5:47 PM
Let me confirm a couple more dates, and then I should have a better idea at my schedule Is that ok?

Emily Braun 6/5 5:48 PM
Yeah, that's completely fine. Just wanting to get ahead of schedule... per usual.

Hi Emily. Here's the updated conference schedule for FY 19 and 20. Let me know if you have any questions. Thanks!

2019-20 Conference Schedule.xlsx
Anyone with the link can edit

Figure 4: Microsoft Teams Interface

To maintain a similar and understandable look and feel of the interface it was determined that the interface should incorporate the most relevant features from this list:

- Clear separation of different user messages through left and right positioning, message bubbles and profile picture or usernames attached to the message
- Limited size of each message
- Message sent times
- Input controls at the bottom of the window

Features such as emojis and support for images could be added in a future iteration of the chatbot but were deemed ultimately unnecessary for the use case. A mock-up of the desktop chat interface produced from these features at the start of the project is shown below in Figure 5.

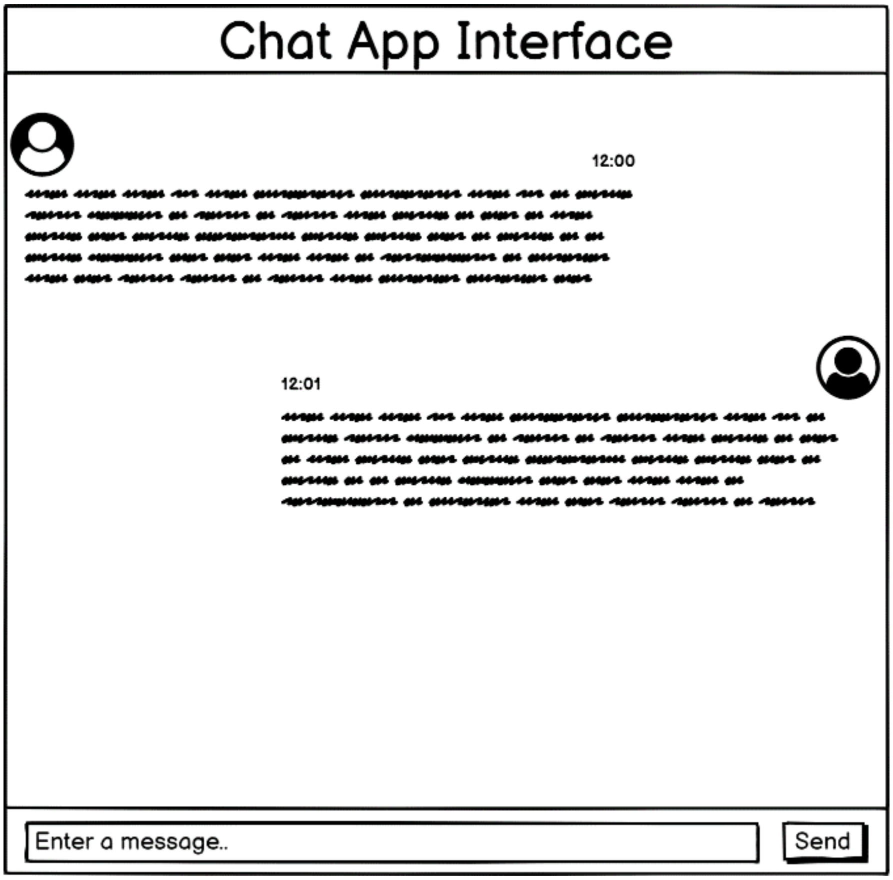
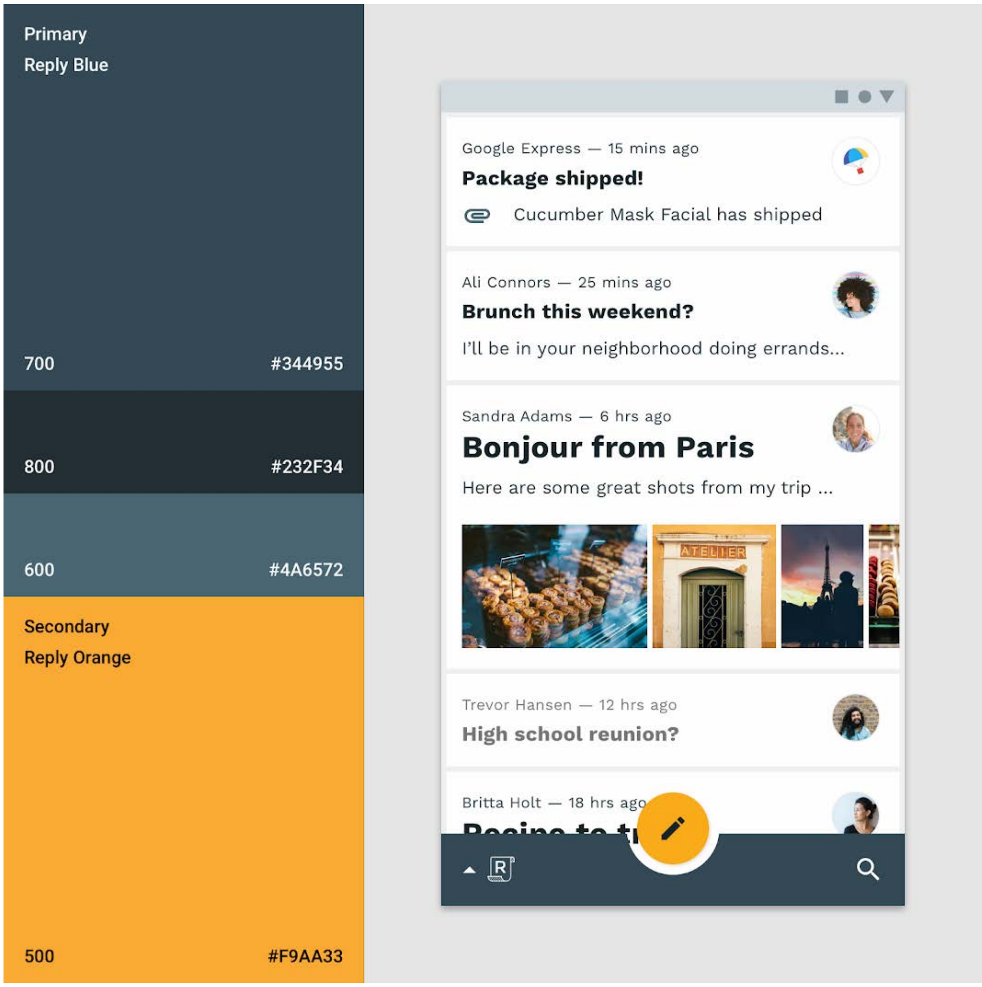


Figure 5: Original Interface Mock-up

After the layout of the interface and the primary features had been decided research into best practices of interface theming and appropriate colour combinations was done. Google's Material Design (CITE) proved to be the best resource for this as it provided both an in-depth study of colour combinations and theming, and a vast library of available fonts and icons to be used freely. The final interface colour scheme was chosen from an adaptation of the Material Design case study 'Reply' (Material Design, 2021)



Google Express — 15 mins ago
Package shipped!
Cucumber Mask Facial has shipped

Ali Connors — 25 mins ago
Brunch this weekend?
I'll be in your neighborhood doing errands...

Sandra Adams — 6 hrs ago
Bonjour from Paris
Here are some great shots from my trip ...

Trevor Hansen — 12 hrs ago
High school reunion?

Britta Holt — 18 hrs ago

Figure 6: Reply Case Study Colours (Material Design, 2021)

The primary grey-blue colour combined with a bright orange secondary colour is intended to be suggestive of the UK rail network. With mostly grey and monotonous trains and stations mixed with bright splashes of colour, such as the vivid orange tickets.

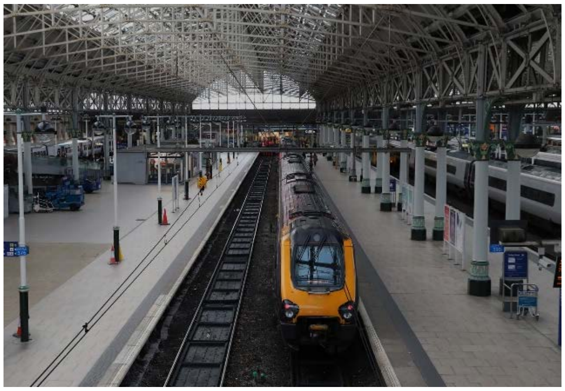
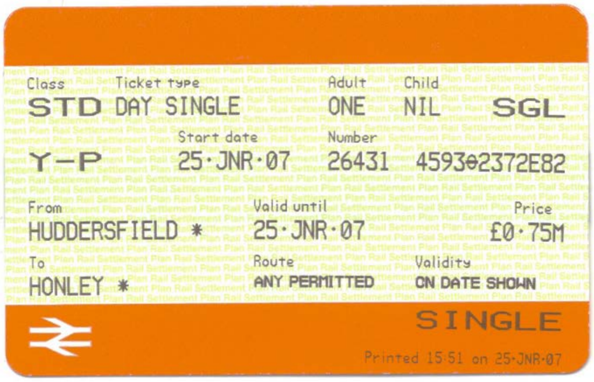


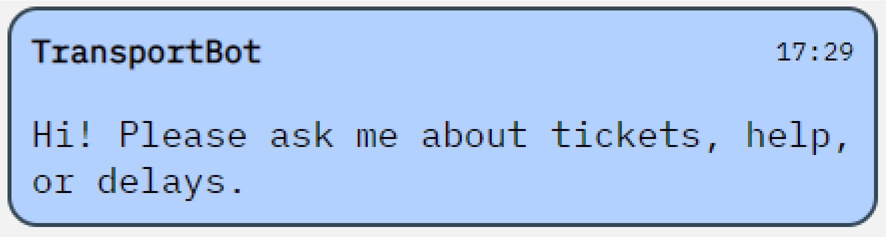
Figure 6: Manchester Piccadilly Station (Livesey, n.d)



Class STD
Ticket type DAY SINGLE
Adult ONE
Child NIL
SGL
Y-P
Start date 25·JNR·07
Number 26431 459302372E82
From HUDDERSFIELD *
Valid until 25·JNR·07
Price £0·75M
To HONLEY *
Route ANY PERMITTED
Validity ON DATE SHOWN
SINGLE
Printed 15:51 on 25·JNR·07

Figure 7: UK Train Ticket (Adambro, 2007)

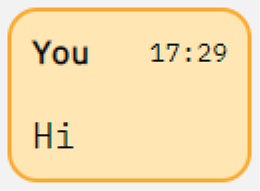
The chat messages sent by both chatbot and user also incorporate lighter versions of the primary and secondary colours, respectively, to further distinguish them from each other.



TransportBot 17:29

Hi! Please ask me about tickets, help, or delays.

Figure 8: Bot Message Bubble



You 17:29

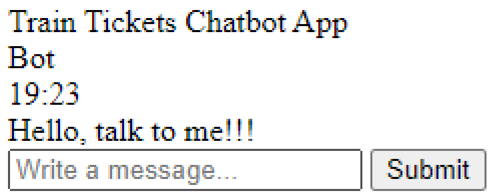
Hi

Figure 9: User Message Bubble

The icons used for the buttons of the interface were sourced from the Material Design resource, and the font for all the text is sourced from Google Fonts (Abbink, n.d).

First Implementation

Very early in development, the first iteration of the interface had only basic functionality and almost no style or design work had been done. The functionality was entirely Flask based and messages were sent to the server with a POST request from the HTML form. This was less than ideal for messaging, however, as the page required refreshing to update the messages in the chat window and mainly served as a starting point to build upon.



Train Tickets Chatbot App

Bot

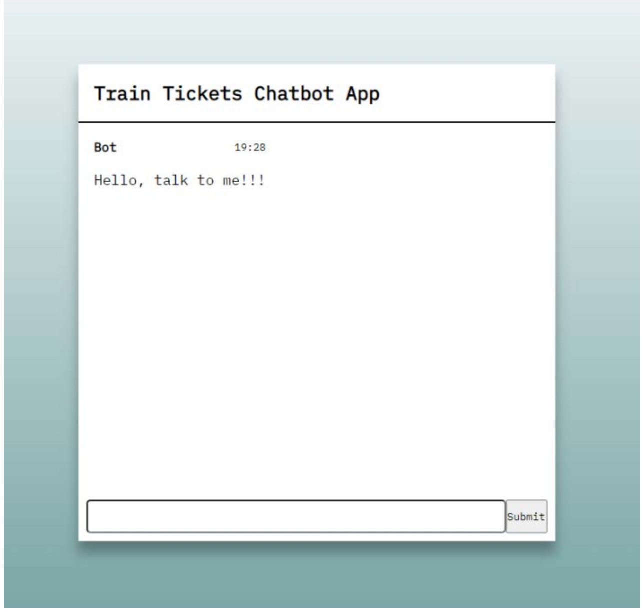
19:23

Hello, talk to me!!!

Figure 10: First Implementation

Second Implementation

The second iteration attempted to focus on implementing the design choices of the interface, with additional work to the HTML layout and incorporation of a CSS stylesheet, it began to resemble initial mock-up. Functionality was unchanged from the initial implementation.



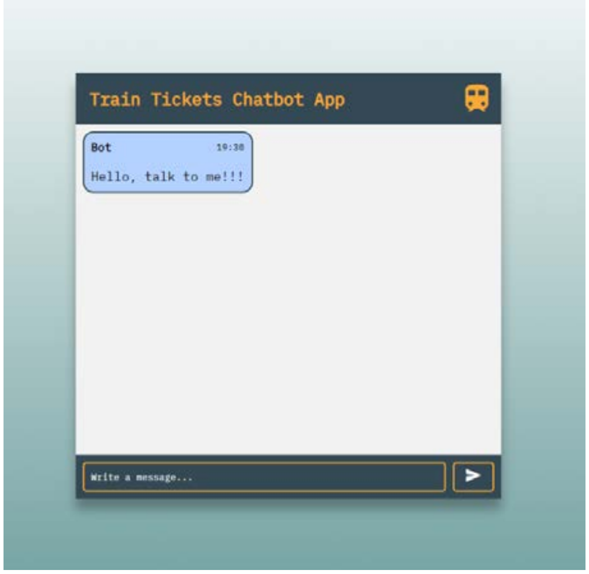
Bot 19:28

Hello, talk to me!!!

Figure 11: Second Implementation

Third Implementation

The third iteration further develops the style of the application, incorporating the selected colour scheme, icons, and message bubbles. To allow messages to be shown in the chat window without refreshing the page, message implementation was moved from being form-based with Flask to using JavaScript to send Fetch requests to the Flask server that contained the user's message, which the chatbot would then respond to inside the Fetch request's response. This implementation incorporated all base features of the application and was fully functional. The major flaw of this implementation, however, was that the messages were sent in a 1:1 manner. The bot could only respond when a user had sent it a message. This was not ideal for general use and made situations where the bot would be required to send multiple messages impossible.



Bot 19:30

Hello, talk to me!!!

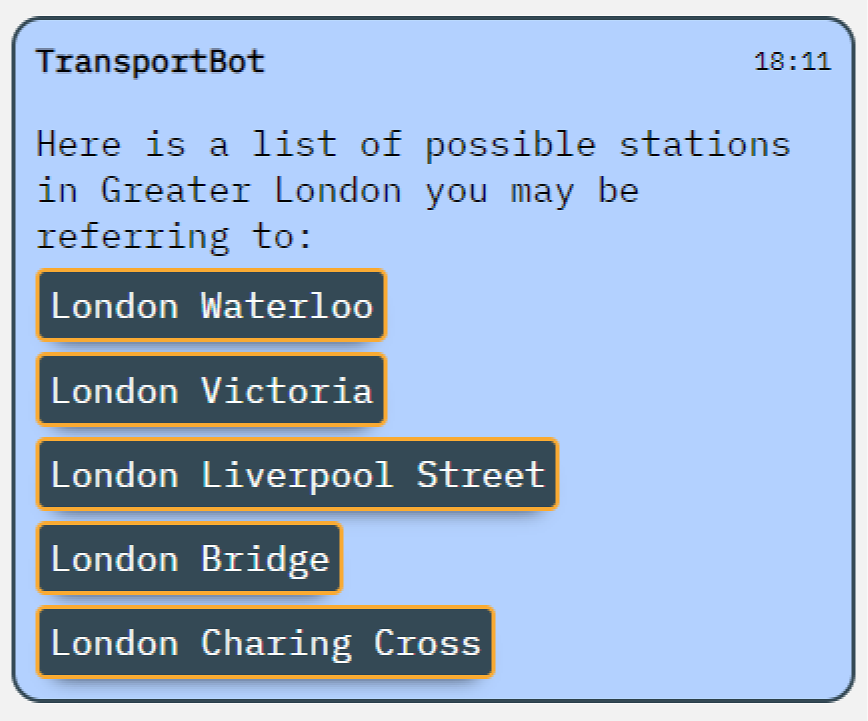
Figure 12: Third Implementation

Final Implementation

The final iteration of the client changed the background of the page to the 'Contact Us' page of the train company Greater Anglia's website (CITE). The choice of company was purely for convenience and is only intended to demonstrate the application closer to the intended use case. The functionality of sending and receiving messages was reworked to use a new socket-based method using the JavaScript library Socket.IO (Piedrafita et al, 2018), and the corresponding server integration facilitated by the Flask-SocketIO library (Grinberg, 2014). This now allowed the messages to be completely independent, allowing the bot to now send messages as and when it needed to, including multiple messages if required, as well as greatly simplifying the underlying functions.

Three additional features were also added: Button lists, Text-to-Speech, and Speech-to-Text.

Button lists



TransportBot 18:11

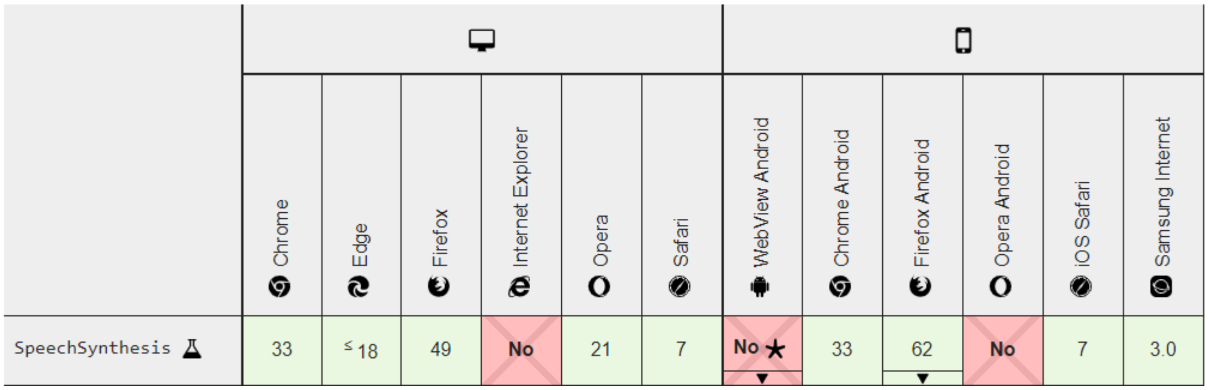
Here is a list of possible stations in Greater London you may be referring to:

Figure 13: London Stations Button List

When the bot needs to ask the user to choose from a list of information, such as in the case of a location-based search of train stations shown above in Figure 13, each element of the list of choices presented to the user is displayed in the form of a list of buttons the user can then press rather than manually input their response. When a button is pressed the button's corresponding choice is immediately sent as a user message back to the bot.

Text-to-speech (TTS)

To improve support for users that either do not wish to, or have difficulty reading the messages sent to them by the bot, support for TTS was added which will read out every message sent by the bot if the user has enabled this setting. This setting is off by default. When the client receives a message from the bot, the client JavaScript checks if the user has enabled TTS. If this is the case then all following messages sent by the bot are first stripped of any unnecessary text, specifically HTML code used for message formatting and the displaying of links and buttons using REGEX. The resulting message is then read aloud using the Speech Synthesis interface of the Web Speech API. This interface is regarded as ‘experimental’ by Mozilla but is shown to be supported by current versions of all major desktop browsers, including Chrome, Edge, Firefox and Safari. Android and IOS mobile web browsers are also supported.

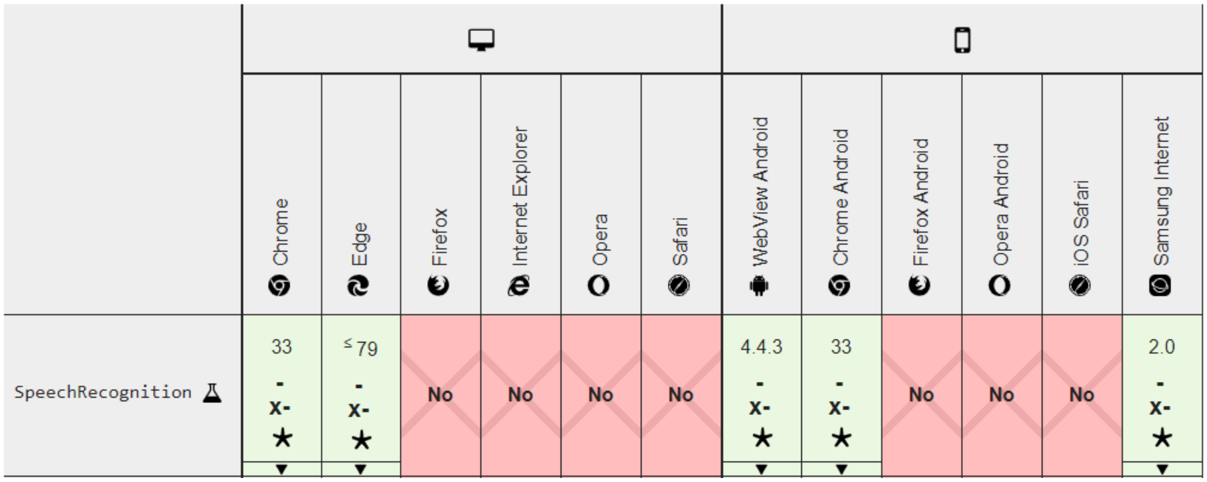


	🖥						📱					
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox Android	Opera Android	iOS Safari	Samsung Internet
`SpeechSynthesis`	33	≤18	49	No	21	7	No ★	33	62	No	7	3.0

Figure 14: Web Speech API SpeechSynthesis Support (Mozilla, 2021)

Speech-to-text

The Web Speech API also has an interface for Speech Recognition, however, this was not used as it is not as well supported. Currently this is only supported by Chrome and Edge desktop browsers as well as only Chrome and Samsung browsers on mobile.

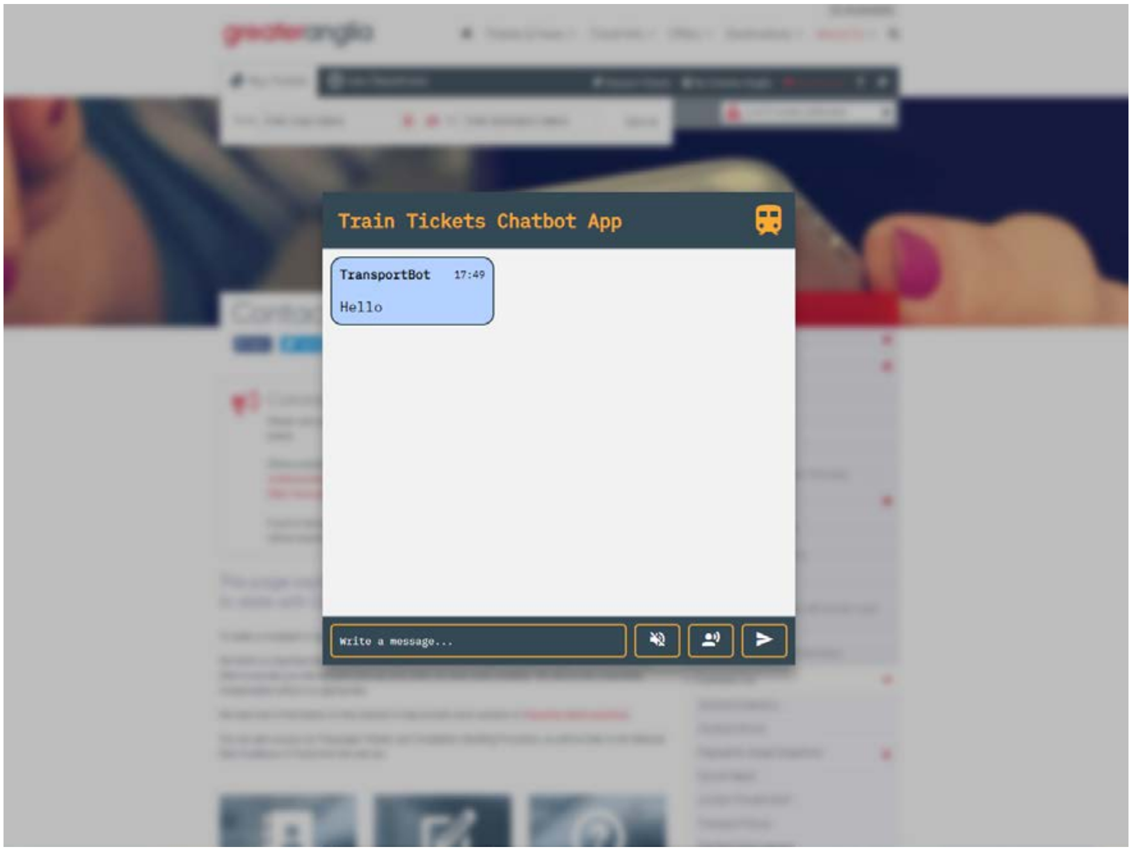


	🖥						📱					
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox Android	Opera Android	iOS Safari	Samsung Internet
`SpeechRecognition`	33 - x- ★	≤79 - x- ★	No	No	No	No	4.4.3 - x- ★	33 - x- ★	No	No	No	2.0 - x- ★

Figure 15: Web Speech API SpeechRecognition Support (Mozilla 2021)

To maximise support for this feature, the user audio is instead processed on the Flask server. First the user’s audio is recorded using the JavaScript library RecordRTC (Khan, 2019), primarily to enable recording in .wav format, before being sent to the Flask server using a Fetch request containing the audio data. The server then processes the audio using the Python SpeechRecognition library (Zhang, 2017) and returns the text processed from the user’s audio in the Fetch response. The client then adds the received text to the message input box for the user to confirm before sending.

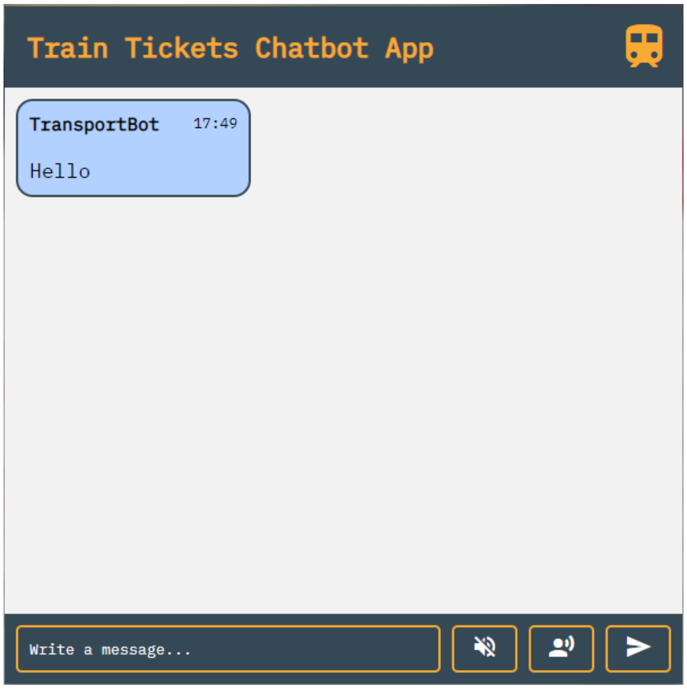
Figures 16, 17 and 18 below show the final version of the client in both desktop and mobile views.



TransportBot 17:49

Hello

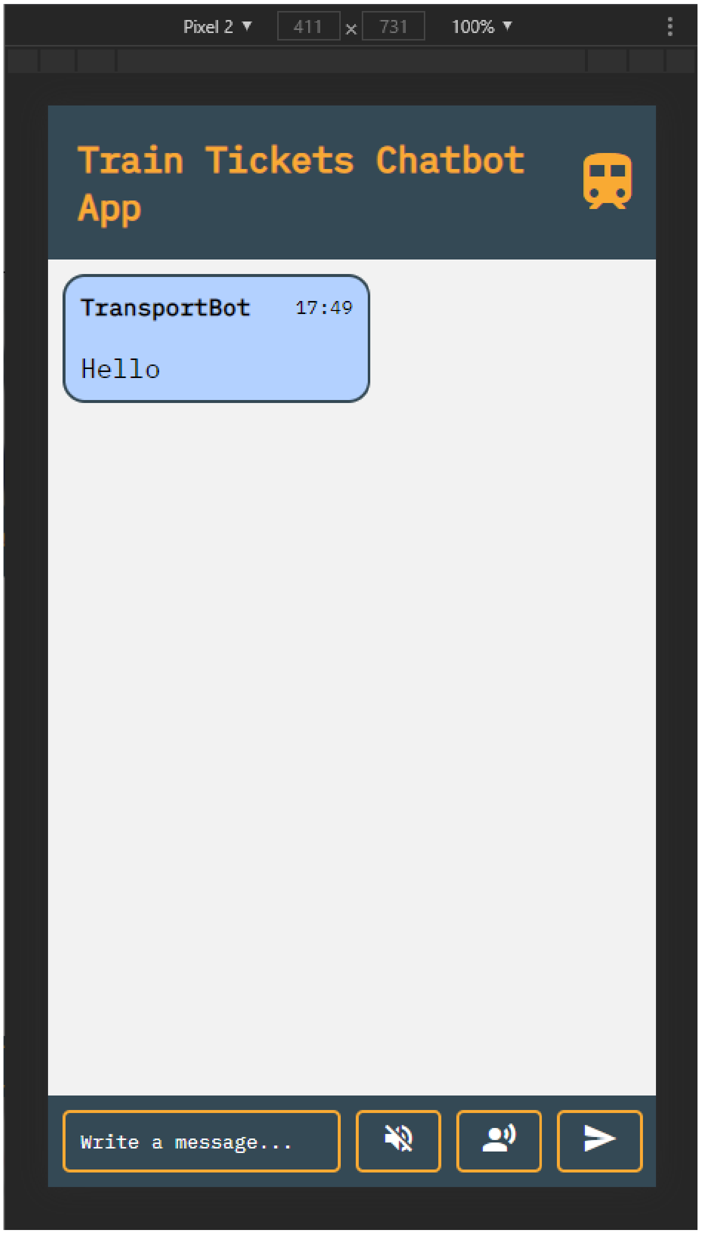
Figure 16: Final Iteration with Background Image



TransportBot 17:49

Hello

Figure 17: Final Iteration Desktop Chat Window



Train Tickets Chatbot App

TransportBot 17:49

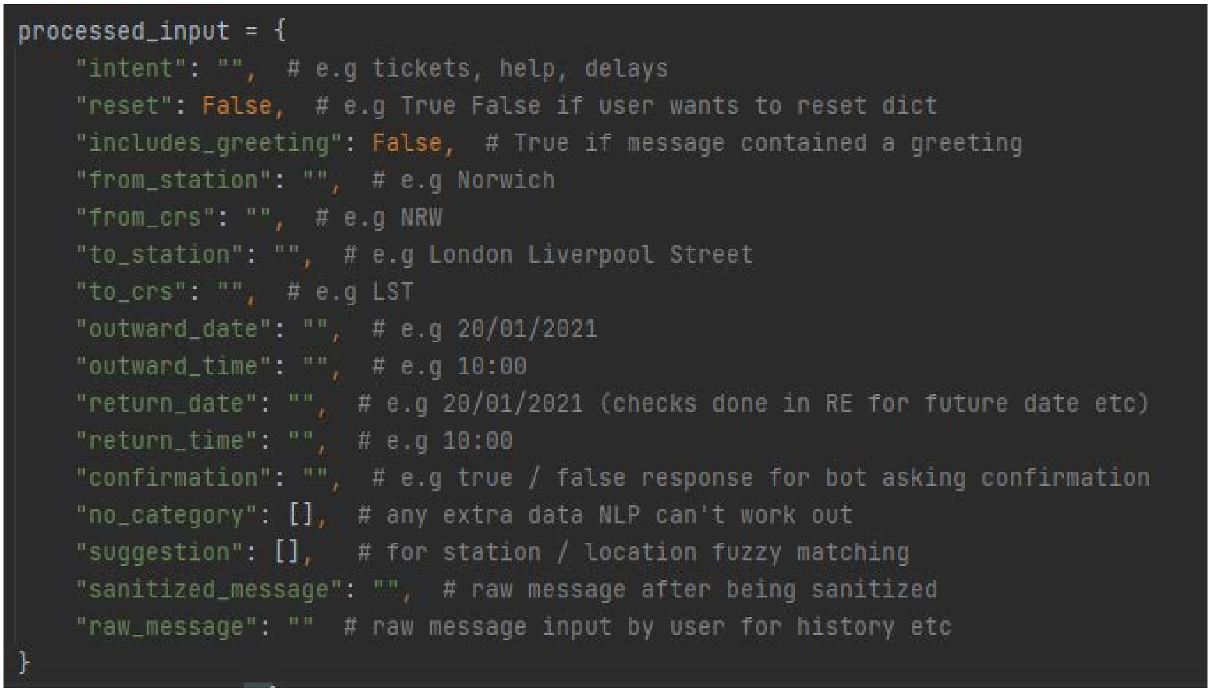
Hello

Figure 18: Final Iteration Mobile View

NLP

Design

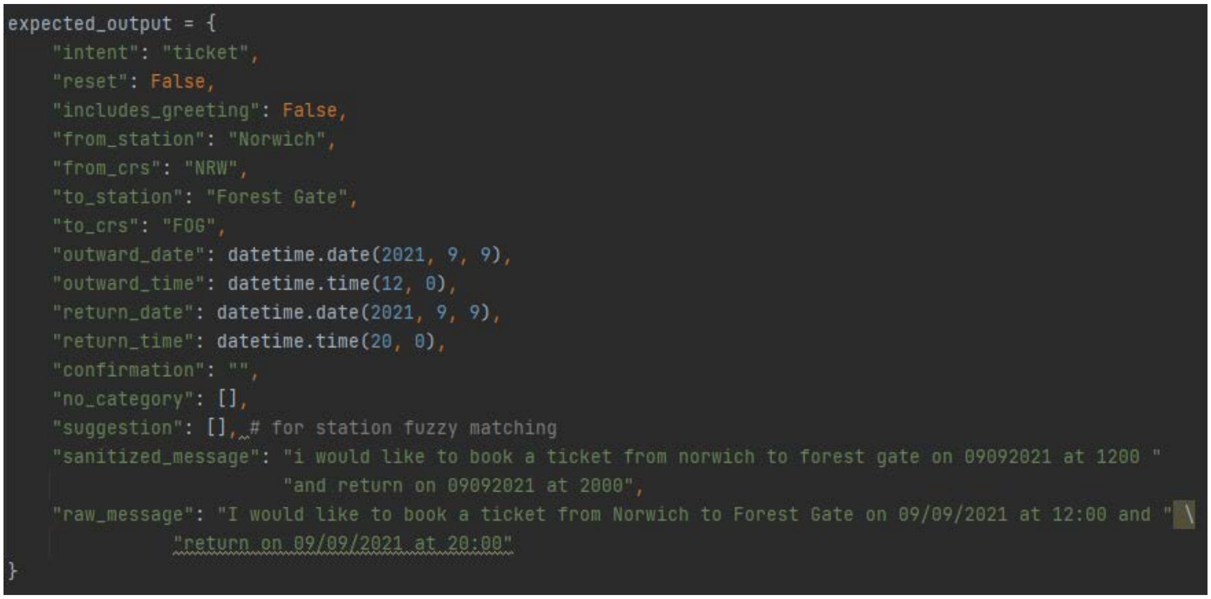
Because the NLP function does not track the state of the main program it is required to try to extract as much relevant information as possible from any given user input as it does not have any specific information to look for. This extracted information is stored in a 16-key dictionary. Since Experta stores its 'Facts' internally as dictionary objects then this allows the Reasoning Engine (RE) to use this provided dictionary to look for specific information at any given stage of a user conversation and make decisions based upon it. The default dictionary can be seen below in Figure 19.



```
processed_input = {
    "intent": "",  # e.g tickets, help, delays
    "reset": False,  # e.g True False if user wants to reset dict
    "includes_greeting": False,  # True if message contained a greeting
    "from_station": "",  # e.g Norwich
    "from_crs": "",  # e.g NRW
    "to_station": "",  # e.g London Liverpool Street
    "to_crs": "",  # e.g LST
    "outward_date": "",  # e.g 20/01/2021
    "outward_time": "",  # e.g 10:00
    "return_date": "",  # e.g 20/01/2021 (checks done in RE for future date etc)
    "return_time": "",  # e.g 10:00
    "confirmation": "",  # e.g true / false response for bot asking confirmation
    "no_category": [],  # any extra data NLP can't work out
    "suggestion": [],   # for station / location fuzzy matching
    "sanitized_message": "",  # raw message after being sanitized
    "raw_message": ""  # raw message input by user for history etc
}
```

Figure 19: NLP Empty Dictionary

For example, in the query, 'I would like to book a ticket from Norwich to Forest Gate on 2021/09/09 at 12:00 and return on 2021/09/09 at 20:00,' would produce the dictionary seen below in Figure 20.



```
expected_output = {
    "intent": "ticket",
    "reset": False,
    "includes_greeting": False,
    "from_station": "Norwich",
    "from_crs": "NRW",
    "to_station": "Forest Gate",
    "to_crs": "FOG",
    "outward_date": datetime.date(2021, 9, 9),
    "outward_time": datetime.time(12, 0),
    "return_date": datetime.date(2021, 9, 9),
    "return_time": datetime.time(20, 0),
    "confirmation": "",
    "no_category": [],
    "suggestion": [], # for station fuzzy matching
    "sanitized_message": "i would like to book a ticket from norwich to forest gate on 09092021 at 1200 "
                         "and return on 09092021 at 2000",
    "raw_message": "I would like to book a ticket from Norwich to Forest Gate on 09/09/2021 at 12:00 and " \
              "return on 09/09/2021 at 20:00"
}
```

Figure 20: Output NLP Dictionary

This approach allows multiple steps to be taken at once by the user, including an entire query if all information has been provided correctly.

Dictionary Details

Key: “intent”

Values: “ticket”, ”help”, ”change” ”cancel”

Description: The user’s primary intent. Usually only established at the beginning of a conversation.

Key: “reset”

Values: True, False

Description: A special intent used to reset and remove all knowledge the chatbot may have of the current conversation.

Key: “includes_greeting”

Values: True, False

Description: A special intent used to determine if the user message contains a greeting.

Key: “from_station”

Values: Any valid station from the database

Description: Used if the user enters a valid station and there is no doubt that this is not the destination station.

Key: “from_crs”

Values: Any valid crs code from the database that matches a station

Description: The matching crs code for any station in the database. Works as a pair with the station key.

Key: “to_station”

Values: Any valid station from the database

Description: Used if the user enters a valid station and there is no doubt that this is the destination station.

Key: “to_crs”

Values: Any valid crs code from the database that matches a station

Description: The matching crs code for any station in the database. Works as a pair with the station key.

Key: “outward_date”

Values: Any valid date

Description: Contains a date only if a valid date is entered and it is certain to be the outgoing journey.

Key: “outward_time”

Values: Any valid time

Description: Contains a time only if a valid date and time is entered and it is certain to be the outgoing journey.

Key: “return_date”

Values: Any valid date

Description: Contains a date only if a valid date is entered and it is certain to be the return journey.

Key: “return_time”

Values: Any valid time

Description: Contains a date only if a valid date and time is entered and it is certain to be the return journey.

Key: “confirmation”

Values: True, False

Description: Used to determine a yes/no response by the user.

Key: “no_category”

Values: Any station, date, or time that is valid but the details are uncertain for

Description: Used to extract information when the details are not clear. For example, in the case of single statement input since NLP does not maintain any knowledge of the state.

Key: “suggestion”

Values: A key value pair where the key is either “station” or “location” and the values are either any valid station from the database, or any valid location from the database

Description: Used by the fuzzy matching function in the case of invalid input to try to determine the best guess to the user’s intent.

Key: “sanitized_message”

Values: A lowercase text string containing the user message with no punctuation and spelling mistakes resolved

Description: Processed user input to remove all invalid characters and spelling mistakes. Used mostly for historical purposes.

Key: “raw_message”

Values: Raw user message

Description: Raw input from the user before any processing

Techniques and Tools

Spacy

Spacy (Spacy.io, 2021) is a popular and well-regarded Natural Language Processing toolkit designed for production use that quickly performs a variety of techniques, including Tokenization, Parts-of-speech Tagging, Named Entity Recognition (NER), Lemmatization, Rule-based Matching, and more. It is the primary tool used by the NLP functionality of the chatbot. In this application, it is primarily used for its pattern matching and dependency parsing capabilities.

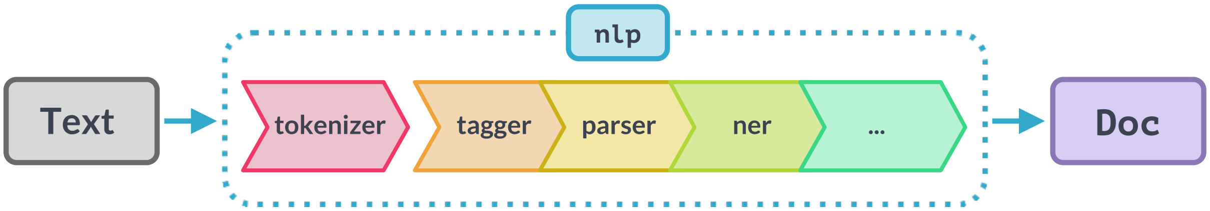


Figure 21: Spacy Processing Pipeline

Spacy's PhraseMatcher allows individual tokens or groups of tokens containing specific text to be found inside a given processed Spacy document. This allows the NLP to search for specific information. For example, if the user enters: "I want to travel to london", a rule specifying user intent could match the word "travel" and deduce the user wants to buy a ticket. This is effective for matching key words that will not overlap but complex queries require more processing.

Spacy parts-of-speech tagging and dependency parsing allows more complex rules and extraction of information by determining the relationship between specific tokens. This is visualised in Figure (X) below for the previous example phrase: "I want to travel to London" and from this it can be seen that "tomorrow" is a dependency of "travel", or that travel is the "head" token for "tomorrow". This relationship allows the NLP functions to work out which parts of the information are connected, and in this example an outgoing journey date of tomorrow's date would be set with the destination of London.

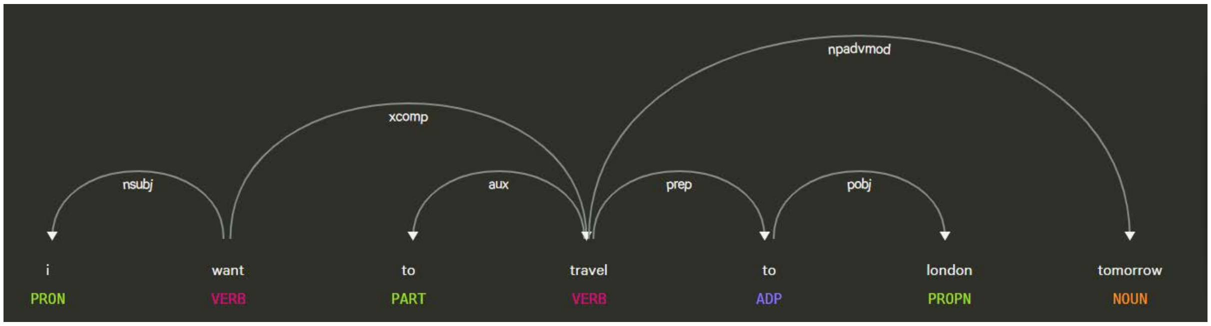


Figure 22: Displacy Dependancy Visualisation

Spacy's Lemmatizer was also used to perform lemmatization on the user input to reduce complexity of the implementation of the PhraseMatcher. This reduces the variances on matched words to their base form. For example, "returning" becomes "return". Spacy's PhraseMatcher does allow lemmatization to be performed while matching, however, when this option was used performance suffered greatly. A run of 24 unit tests took 545.141s with this option enabled and only 20.796s with it disabled. A solution was to pre-process the user input using Spacy's lemmatizer and running the PhraseMatcher on that instead. This reduced the run time to 33.901s.

Named Entity Recognition was not used as this was not able to function for single word entry use-cases.

Fuzzywuzzy

Fuzzywuzzy (Gareth, et al, 2020) performs fuzzy matching using Levenshtein Distance to find similarities between different text sources. This is used to try to match incorrectly entered station names or locations by comparing the entered words with the known stations and locations from the database.

Datefinder

Datefinder (Koumijan, 2015) is a python library used to create datetime objects from textual dates. It is used to extract all valid dates from the user input. Due to an implementation problem in datefinder certain dates are returned by default in US format and for relative dates such as "today" and "tomorrow" must be first processed into DD/MM/YYYY format to compensate for this.

Spellchecker

Spellchecker (Barrus, 2018) is a python library used to correct common spelling errors in English language text sources. Additional words can be added to its vocabulary, allowing stations and locations from the database to be correctly fixed. Combined with fuzzy matching this further increases the chances that errors can be accounted for.

Implementation Flowchart

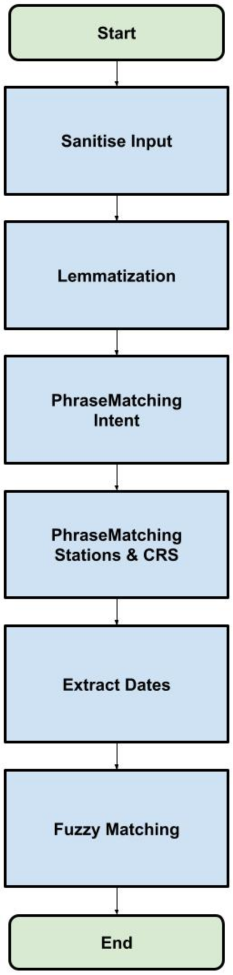
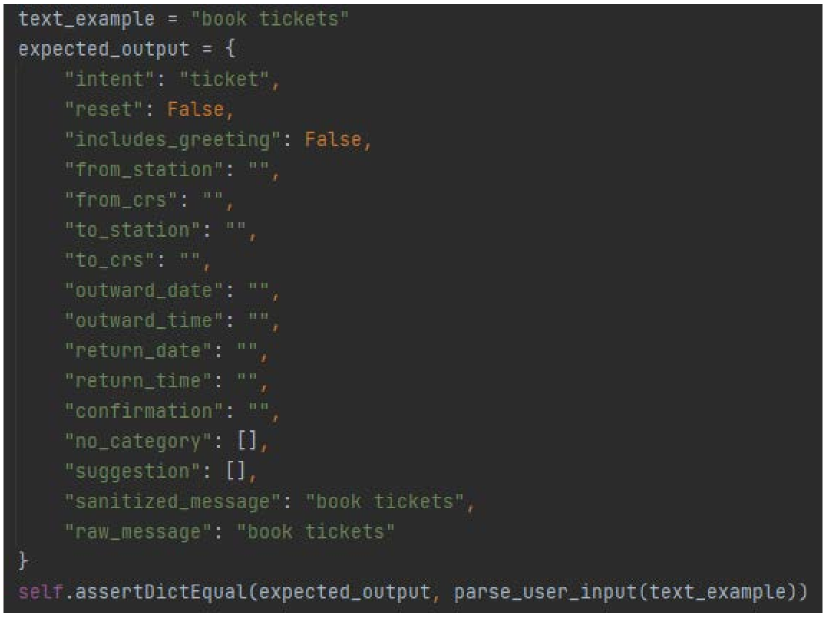


Figure 23: Flowchart of NLP's Main Process

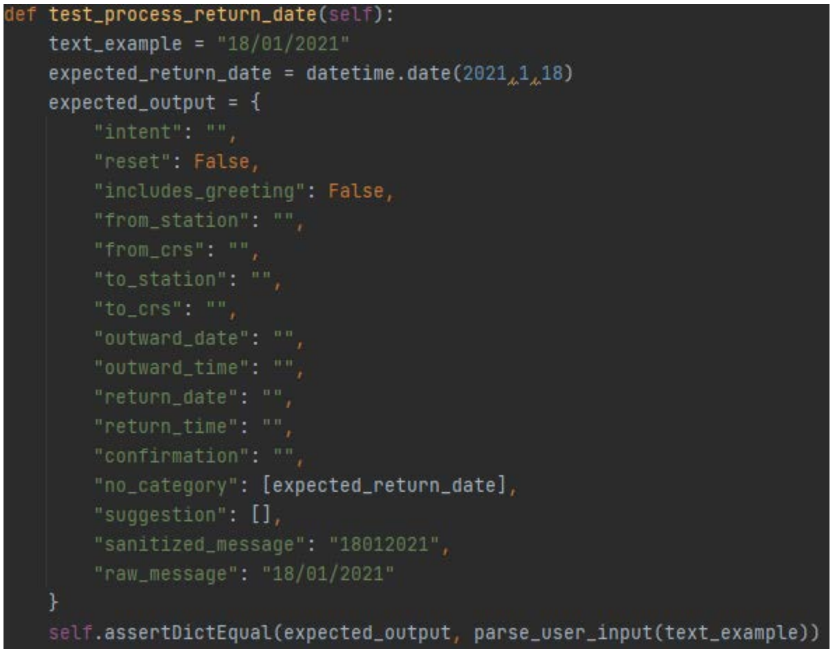
Testing

Because the required output was known ahead of time NLP was perfect for a test-driven development style approach. Unit tests were made with the required output ahead of the required code. After every major change or additional feature in development, all tests were re-run and any problems could be detected immediately. Twenty-four unit tests were used in developing NLP.py and can be found in test_nlp.py. A selection of four unit tests are shown below as examples.



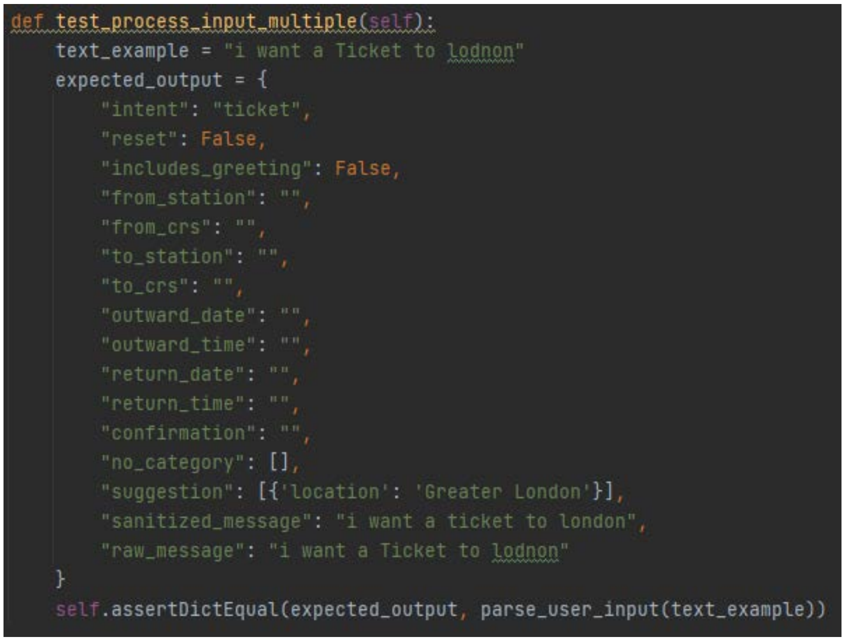
```
text_example = "book tickets"
expected_output = {
    "intent": "ticket",
    "reset": False,
    "includes_greeting": False,
    "from_station": "",
    "from_crs": "",
    "to_station": "",
    "to_crs": "",
    "outward_date": "",
    "outward_time": "",
    "return_date": "",
    "return_time": "",
    "confirmation": "",
    "no_category": [],
    "suggestion": [],
    "sanitized_message": "book tickets",
    "raw_message": "book tickets"
}
self.assertDictEqual(expected_output, parse_user_input(text_example))
```

Figure 24: Unit Test for User Intent



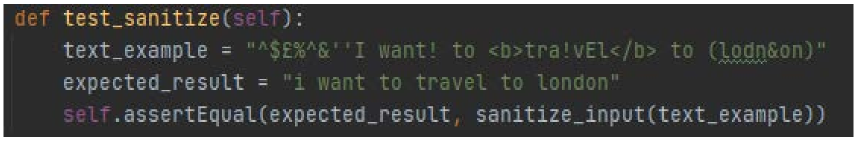
```
def test_process_return_date(self):
    text_example = "18/01/2021"
    expected_return_date = datetime.date(2021,1,18)
    expected_output = {
        "intent": "",
        "reset": False,
        "includes_greeting": False,
        "from_station": "",
        "from_crs": "",
        "to_station": "",
        "to_crs": "",
        "outward_date": "",
        "outward_time": "",
        "return_date": "",
        "return_time": "",
        "confirmation": "",
        "no_category": [expected_return_date],
        "suggestion": [],
        "sanitized_message": "18012021",
        "raw_message": "18/01/2021"
    }
    self.assertDictEqual(expected_output, parse_user_input(text_example))
```

Figure 25: Unit Test for Date Extraction



```python
def test_process_input_multiple(self):
    text_example = "i want a Ticket to lodnon"
    expected_output = {
        "intent": "ticket",
        "reset": False,
        "includes_greeting": False,
        "from_station": "",
        "from_crs": "",
        "to_station": "",
        "to_crs": "",
        "outward_date": "",
        "outward_time": "",
        "return_date": "",
        "return_time": "",
        "confirmation": "",
        "no_category": [],
        "suggestion": [{'location': 'Greater London'}],
        "sanitized_message": "i want a ticket to london",
        "raw_message": "i want a Ticket to lodnon"
    }
    self.assertDictEqual(expected_output, parse_user_input(text_example))
```

Figure 26: Unit Test for Short Query with Errors



```python
def test_sanitize(self):
    text_example = "^$£%^&''I want! to <b>tra!vEl</b> to (lodn&on)"
    expected_result = "i want to travel to london"
    self.assertEqual(expected_result, sanitize_input(text_example))
```

Figure 27: Unit Test for Input Sanitisation

Knowledgebase & Inference Engine

This section of the chat bot handles the user’s response from the NLP, processes the information the NLP has given to generate valid feedback for the user.

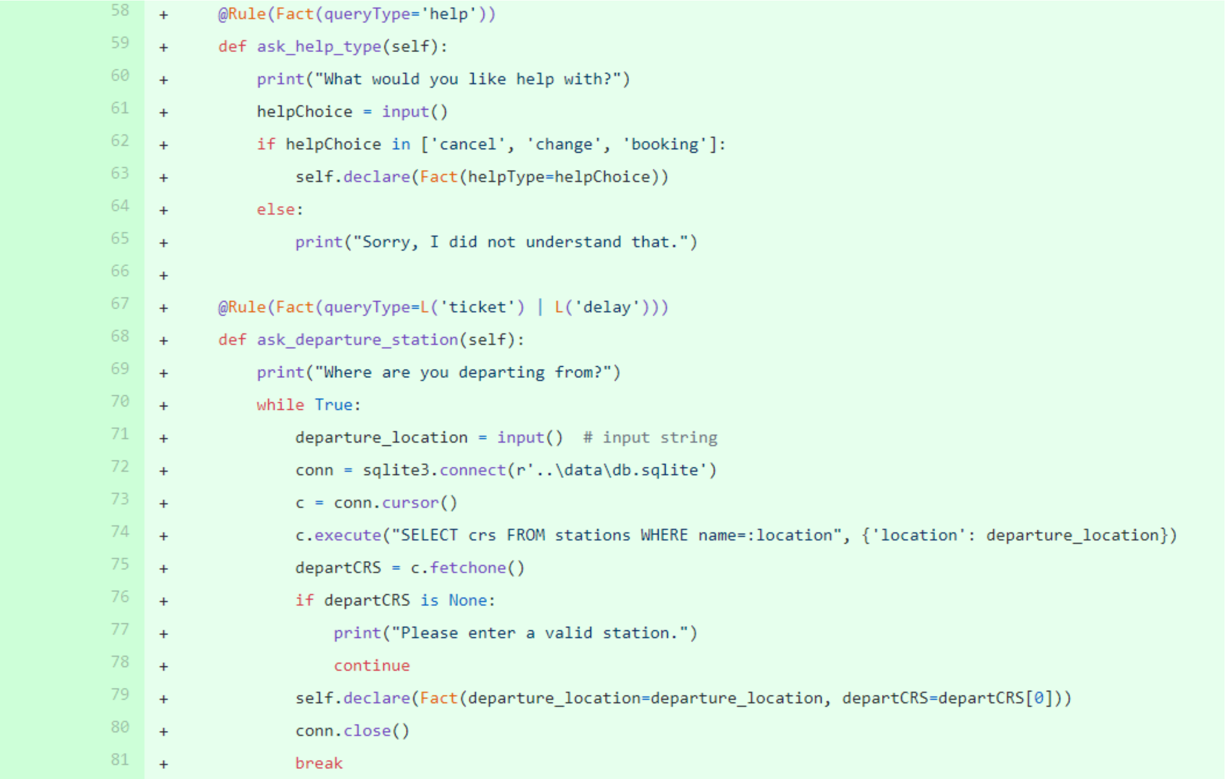
Process

To build a reasoning engine, we must make a choice in which expert system we would like to incorporate into our chat bot. The first option we tired for was PyKe, PyKe is a knowledge engine that is heavily inspired by the language Prolog, in order to fully make use of the engine we must understand how it works and how it will be integrated with the rest of the components. Unfortunately, we were not able to figure out how to use PyKe effectively with the given deadline set for the inference engine, the engine was somewhat difficult to understand due to the language it takes after, and the lack of examples did not help as this was not a very popular engine. However, we’ve learnt the uses of forward and backward chaining and how the inference engine should operate. We then turned to a newer knowledge engine, Pyknow, it was also known as Experta before, we had a lot of success in using Pyknow thus it allowed us to develop a knowledge engine suited towards answering train ticket queries.

It also came apparent that the knowledgebase is built off what the user has entered, thus we consider the knowledgebase and inference engine as one thing. Validation is also used to filter out information that should or should not be in the knowledgebase, if the user enters incorrect information they should then be prompted for the correct input before proceeding to the next query. The inference engine we built only uses forward chaining to derive appropriate answers for the end user as finding delays or tickets is a data driven approach, an example for providing train tickets is to continually query the user to enter appropriate details about their journey until the engine reaches its goal of finding a ticket.

Implementation

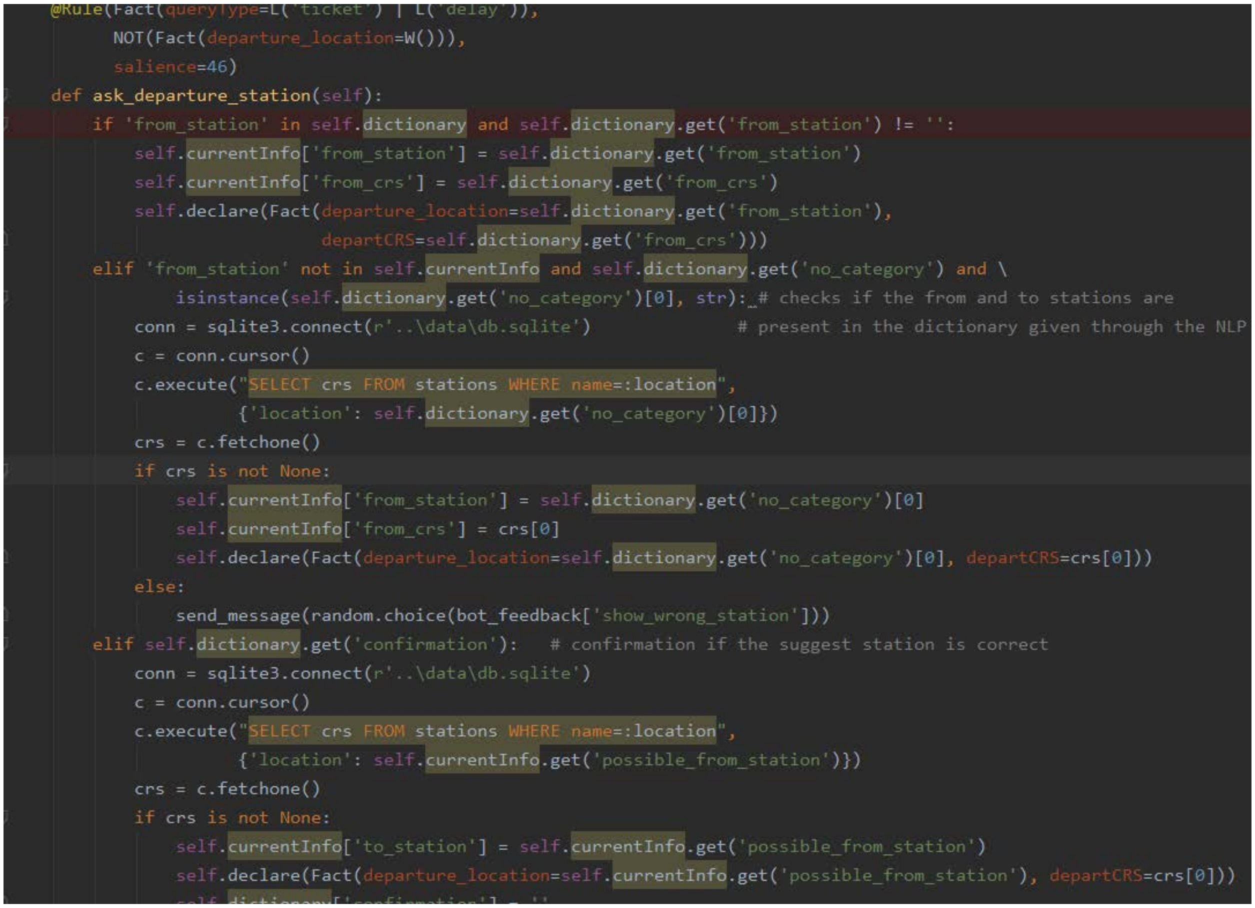
During the early stages of development for the engine, the NLP was not ready to be used so we took the approach of working through the console to build the structure of the chat flow. All rules followed this structure for implementation, prompting us for an input from the user through the console and using a while loop to keep the user there until the correct information is given. The issues with this would be typing out the specific case needed for the engine to run, despite it not being the best solution in developing the engine it allowed us to make placeholders for further changes.



```
+    @Rule(Fact(queryType='help'))
+    def ask_help_type(self):
+        print("What would you like help with?")
+        helpChoice = input()
+        if helpChoice in ['cancel', 'change', 'booking']:
+            self.declare(Fact(helpType=helpChoice))
+        else:
+            print("Sorry, I did not understand that.")
+
+    @Rule(Fact(queryType=L('ticket') | L('delay')))
+    def ask_departure_station(self):
+        print("Where are you departing from?")
+        while True:
+            departure_location = input()  # input string
+            conn = sqlite3.connect(r'..\data\db.sqlite')
+            c = conn.cursor()
+            c.execute("SELECT crs FROM stations WHERE name=:location", {'location': departure_location})
+            departCRS = c.fetchone()
+            if departCRS is None:
+                print("Please enter a valid station.")
+                continue
+            self.declare(Fact(departure_location=departure_location, departCRS=departCRS[0]))
+            conn.close()
+            break
```

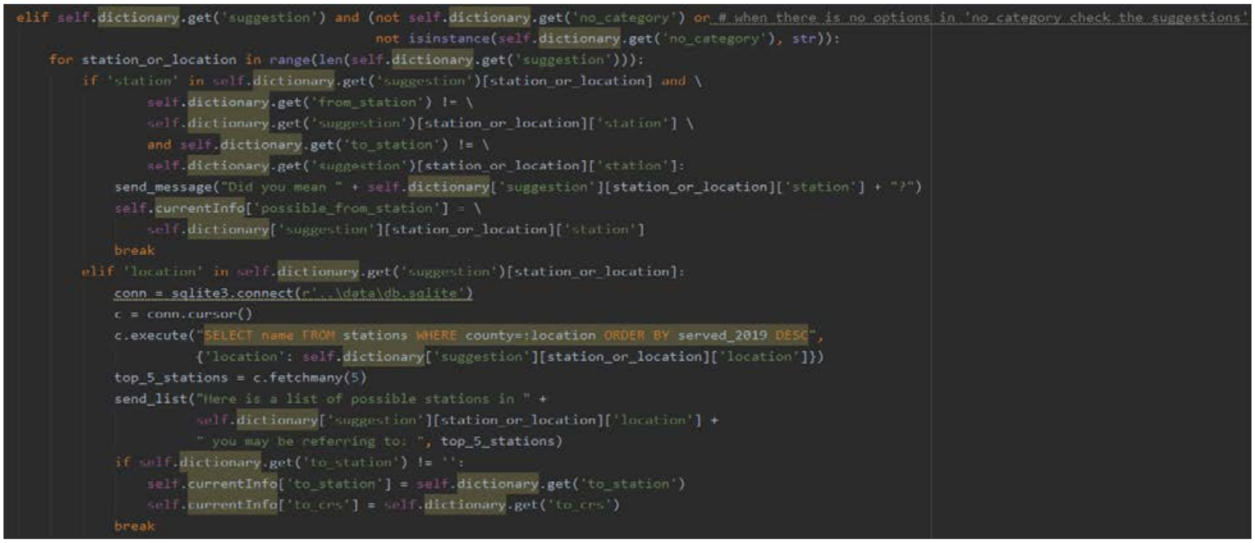
Figure 28: First Iteration, no NLP integrated, use of print statements to get an idea of how the expert system should run

It was also decided that certain validations should be done within the reasoning engine as it is regarded as the center of the bot due to it handling responses from a user; information such as ticket dates, times, and station names should be thoroughly checked for any errors before proceeding to the next query. The NLP was also integrated in figure 20 so the engine does not need a specific string or cases to navigate through the engine, every time a user enters a message the NLP will send a dictionary to the reasoning engine to then digest that response and give back an appropriate feedback to the user.



```
@Rule(Fact(queryType=L('ticket') | L('delay')),
      NOT(Fact(departure_location=W())),
      salience=46)
def ask_departure_station(self):
    if 'from_station' in self.dictionary and self.dictionary.get('from_station') != '':
        self.currentInfo['from_station'] = self.dictionary.get('from_station')
        self.currentInfo['from_crs'] = self.dictionary.get('from_crs')
        self.declare(Fact(departure_location=self.dictionary.get('from_station'),
                          departCRS=self.dictionary.get('from_crs')))
    elif 'from_station' not in self.currentInfo and self.dictionary.get('no_category') and \
            isinstance(self.dictionary.get('no_category')[0], str): # checks if the from and to stations are
        conn = sqlite3.connect(r'..\data\db.sqlite')              # present in the dictionary given through the NLP
        c = conn.cursor()
        c.execute("SELECT crs FROM stations WHERE name=:location",
                  {'location': self.dictionary.get('no_category')[0]})
        crs = c.fetchone()
        if crs is not None:
            self.currentInfo['from_station'] = self.dictionary.get('no_category')[0]
            self.currentInfo['from_crs'] = crs[0]
            self.declare(Fact(departure_location=self.dictionary.get('no_category')[0], departCRS=crs[0]))
        else:
            send_message(random.choice(bot_feedback['show_wrong_station']))
    elif self.dictionary.get('confirmation'):   # confirmation if the suggest station is correct
        conn = sqlite3.connect(r'..\data\db.sqlite')
        c = conn.cursor()
        c.execute("SELECT crs FROM stations WHERE name=:location",
                  {'location': self.currentInfo.get('possible_from_station')})
        crs = c.fetchone()
        if crs is not None:
            self.currentInfo['to_station'] = self.currentInfo.get('possible_from_station')
            self.declare(Fact(departure_location=self.currentInfo.get('possible_from_station'), departCRS=crs[0]))
```

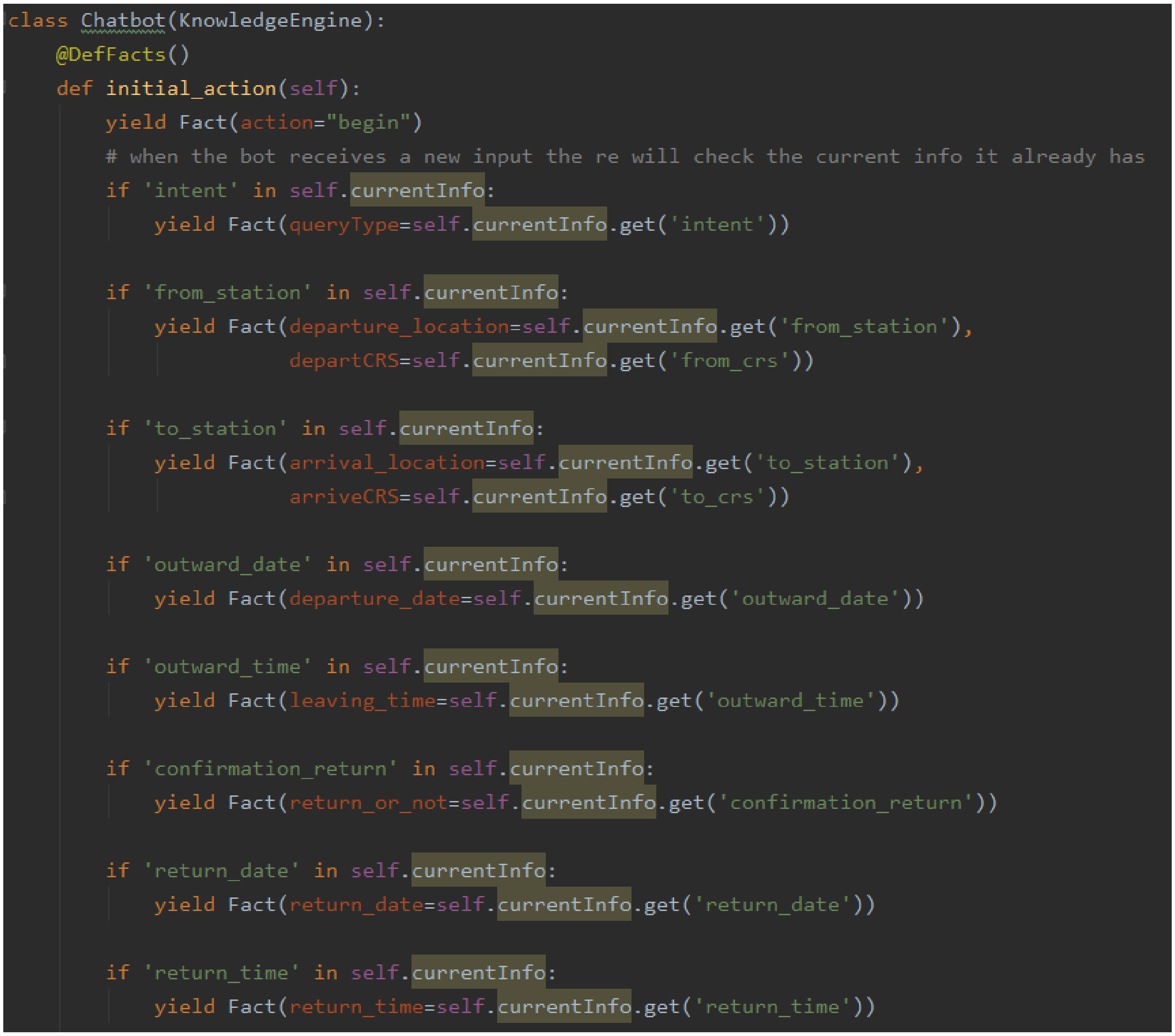
Figure 29: Second Iteration, NLP integrated, using validation to check for the appropriate location



```
elif self.dictionary.get('suggestion') and (not self.dictionary.get('no_category') or # when there is no options in 'no_category check the suggestions'
                                            not isinstance(self.dictionary.get('no_category'), str)):
    for station_or_location in range(len(self.dictionary.get('suggestion'))):
        if 'station' in self.dictionary.get('suggestion')[station_or_location] and \
                self.dictionary.get('from_station') != \
                self.dictionary.get('suggestion')[station_or_location]['station'] \
                and self.dictionary.get('to_station') != \
                self.dictionary.get('suggestion')[station_or_location]['station']:
            send_message("Did you mean " + self.dictionary['suggestion'][station_or_location]['station'] + "?")
            self.currentInfo['possible_from_station'] = \
                self.dictionary['suggestion'][station_or_location]['station']
            break
        elif 'location' in self.dictionary.get('suggestion')[station_or_location]:
            conn = sqlite3.connect(r'..\data\db.sqlite')
            c = conn.cursor()
            c.execute("SELECT name FROM stations WHERE county=:location ORDER BY served_2019 DESC",
                      {'location': self.dictionary['suggestion'][station_or_location]['location']})
            top_5_stations = c.fetchmany(5)
            send_list("Here is a list of possible stations in " +
                      self.dictionary['suggestion'][station_or_location]['location'] +
                      " you may be referring to: ", top_5_stations)
            if self.dictionary.get('to_station') != '':
                self.currentInfo['to_station'] = self.dictionary.get('to_station')
                self.currentInfo['to_crs'] = self.dictionary.get('to_crs')
            break
```

Figure 30: Second Iteration, expert system suggests an appropriate location for the user

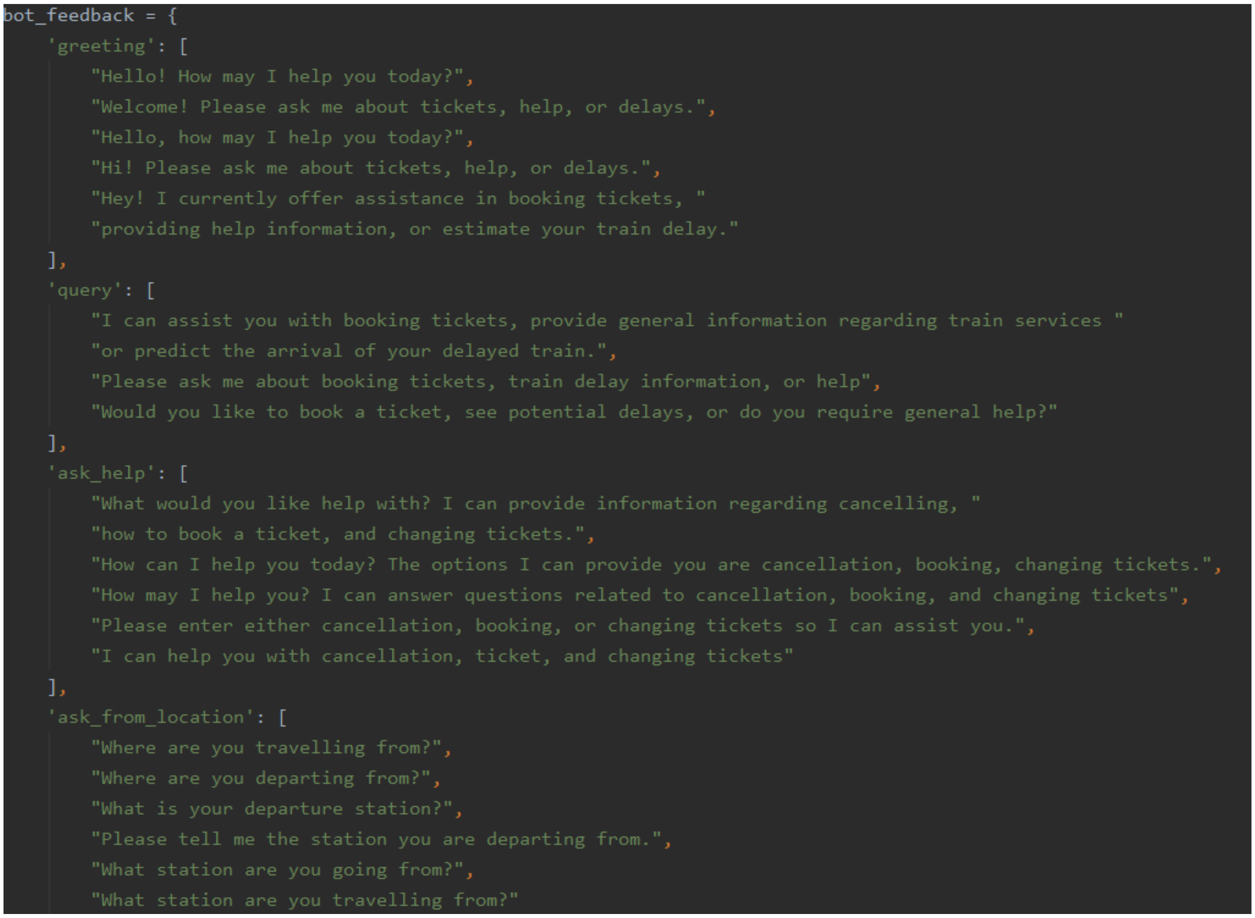
We also found that the knowledgebase can be a separate dictionary and can be built on the fly when querying users, every time the user enters something meaningful in that instance the knowledgebase updates its dictionary so it remembers where the conversation is at appropriately. Figure 22 shows that every time a new dictionary comes from the NLP the current information that was already stored is declared again thus making it seem like the conversation has not stopped.



```
class Chatbot(KnowledgeEngine):
    @DefFacts()
    def initial_action(self):
        yield Fact(action="begin")
        # when the bot receives a new input the re will check the current info it already has
        if 'intent' in self.currentInfo:
            yield Fact(queryType=self.currentInfo.get('intent'))

        if 'from_station' in self.currentInfo:
            yield Fact(departure_location=self.currentInfo.get('from_station'),
                       departCRS=self.currentInfo.get('from_crs'))

        if 'to_station' in self.currentInfo:
            yield Fact(arrival_location=self.currentInfo.get('to_station'),
                       arriveCRS=self.currentInfo.get('to_crs'))

        if 'outward_date' in self.currentInfo:
            yield Fact(departure_date=self.currentInfo.get('outward_date'))

        if 'outward_time' in self.currentInfo:
            yield Fact(leaving_time=self.currentInfo.get('outward_time'))

        if 'confirmation_return' in self.currentInfo:
            yield Fact(return_or_not=self.currentInfo.get('confirmation_return'))

        if 'return_date' in self.currentInfo:
            yield Fact(return_date=self.currentInfo.get('return_date'))

        if 'return_time' in self.currentInfo:
            yield Fact(return_time=self.currentInfo.get('return_time'))
```

Figure 31: Second Iteration, using validation to check for the appropriate location

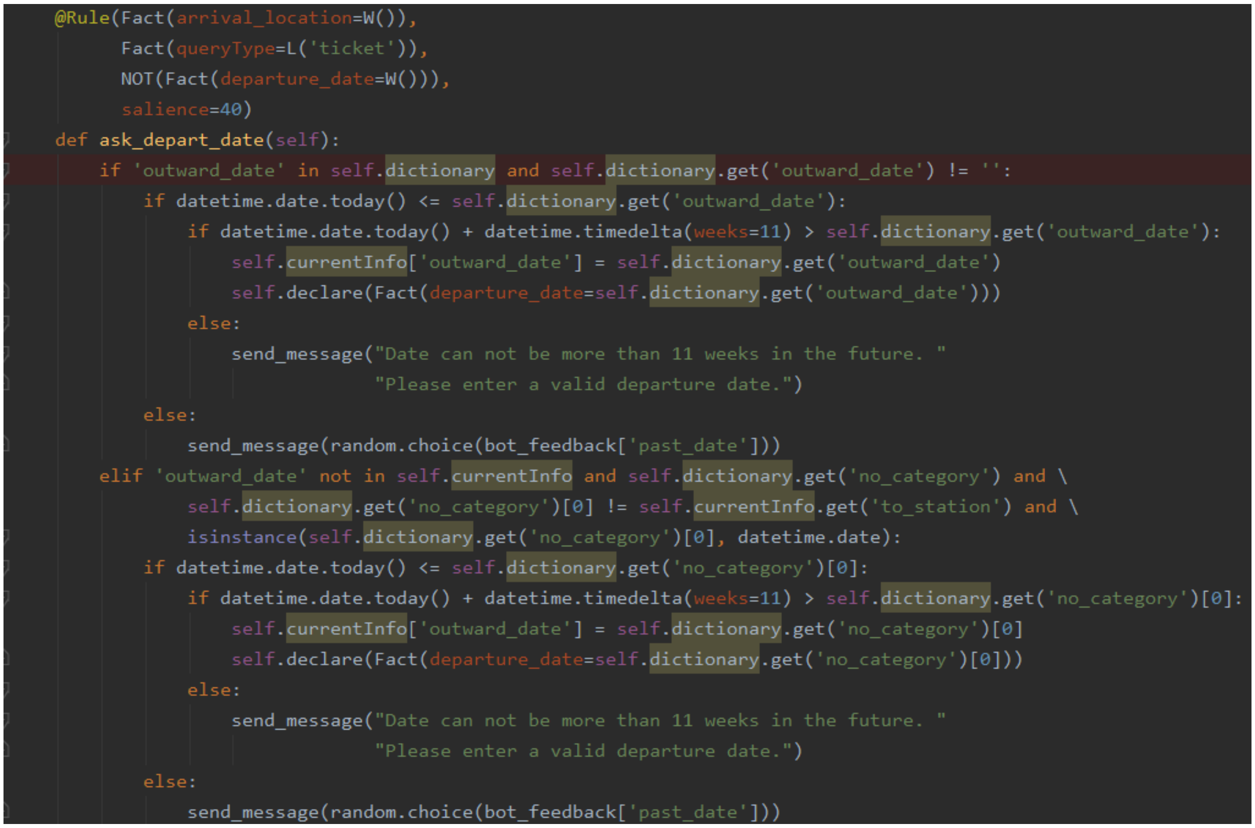
A dictionary of lists of feedback responses was made to incorporate a more natural feel of the conversation so the end user is not prompted with the same dialog. Figure 23 shows some of the possible outcomes for the user.



```
bot_feedback = {
    'greeting': [
        "Hello! How may I help you today?",
        "Welcome! Please ask me about tickets, help, or delays.",
        "Hello, how may I help you today?",
        "Hi! Please ask me about tickets, help, or delays.",
        "Hey! I currently offer assistance in booking tickets, "
        "providing help information, or estimate your train delay."
    ],
    'query': [
        "I can assist you with booking tickets, provide general information regarding train services "
        "or predict the arrival of your delayed train.",
        "Please ask me about booking tickets, train delay information, or help",
        "Would you like to book a ticket, see potential delays, or do you require general help?"
    ],
    'ask_help': [
        "What would you like help with? I can provide information regarding cancelling, "
        "how to book a ticket, and changing tickets.",
        "How can I help you today? The options I can provide you are cancellation, booking, changing tickets.",
        "How may I help you? I can answer questions related to cancellation, booking, and changing tickets",
        "Please enter either cancellation, booking, or changing tickets so I can assist you.",
        "I can help you with cancellation, ticket, and changing tickets"
    ],
    'ask_from_location': [
        "Where are you travelling from?",
        "Where are you departing from?",
        "What is your departure station?",
        "Please tell me the station you are departing from.",
        "What station are you going from?",
        "What station are you travelling from?"
```

Figure 32: Second Iteration, list of different types of responses

Dates can either be categorised in outward_date or no_category or return_date depending on how the NLP interprets it. For more information about the NLP please



```
    @Rule(Fact(arrival_location=W()),
          Fact(queryType=L('ticket')),
          NOT(Fact(departure_date=W())),
          salience=40)
    def ask_depart_date(self):
        if 'outward_date' in self.dictionary and self.dictionary.get('outward_date') != '':
            if datetime.date.today() <= self.dictionary.get('outward_date'):
                if datetime.date.today() + datetime.timedelta(weeks=11) > self.dictionary.get('outward_date'):
                    self.currentInfo['outward_date'] = self.dictionary.get('outward_date')
                    self.declare(Fact(departure_date=self.dictionary.get('outward_date')))
                else:
                    send_message("Date can not be more than 11 weeks in the future. "
                                 "Please enter a valid departure date.")
            else:
                send_message(random.choice(bot_feedback['past_date']))
        elif 'outward_date' not in self.currentInfo and self.dictionary.get('no_category') and \
                self.dictionary.get('no_category')[0] != self.currentInfo.get('to_station') and \
                isinstance(self.dictionary.get('no_category')[0], datetime.date):
            if datetime.date.today() <= self.dictionary.get('no_category')[0]:
                if datetime.date.today() + datetime.timedelta(weeks=11) > self.dictionary.get('no_category')[0]:
                    self.currentInfo['outward_date'] = self.dictionary.get('no_category')[0]
                    self.declare(Fact(departure_date=self.dictionary.get('no_category')[0]))
                else:
                    send_message("Date can not be more than 11 weeks in the future. "
                                 "Please enter a valid departure date.")
            else:
                send_message(random.choice(bot_feedback['past_date']))
```

Figure 33: Final Iteration,

Testing

A unit test could not be done within our test directory, so we decided to test the reasoning engine using people who do not have any prior knowledge of the bot to give their feedback and uncover bugs we did not realise. We also tested it ourselves, creating specific case scenarios for the reasoning engine to handle.

Here is a list of bugs our users have identified and have been fixed:

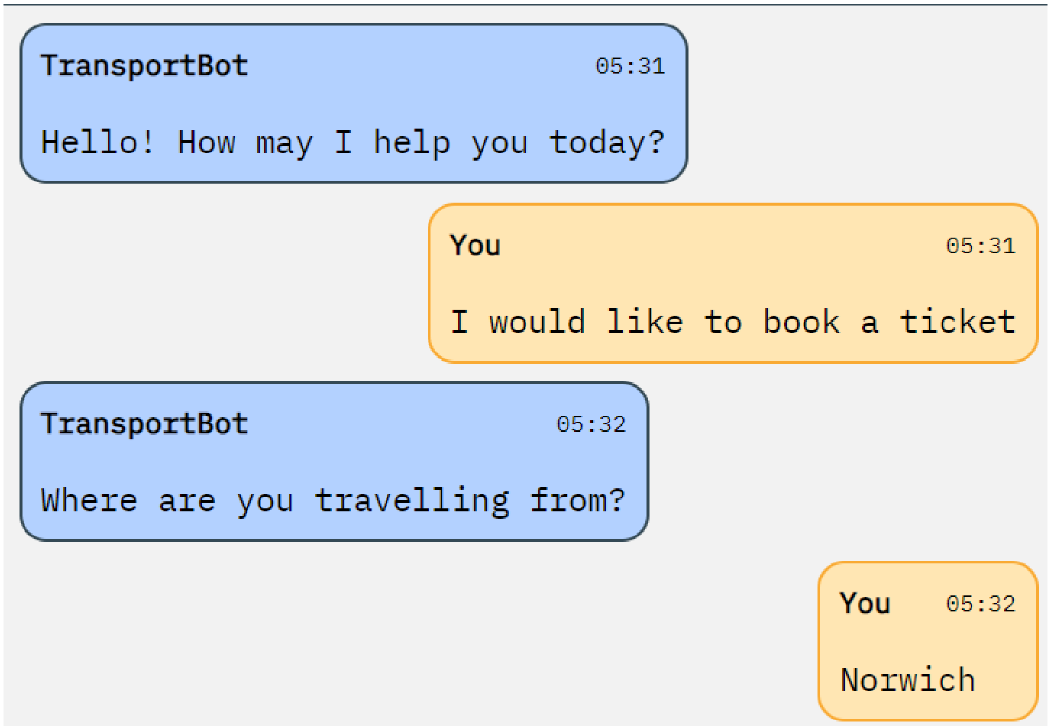
- Bot suggests same location as was typed. I want to travel, London Liverpool street, “Did you mean London Liverpool Street”. Was in no category. Possible to just check case insensitive matching before asking, or word similarity, or compare no category with suggestion
- Bot did not understand yes or no after asking to confirm station / incorrect info “Did you mean London Liverpool Street?”, yes, “I did not understand…”.
- Typing “i want to travel to london liverpool street” does not give any response

- Bot did not pick up on return despite being in dict. “I would like to book a ticket from Norwich to Forest Gate on 2021/09/09 at 12:00 and returning on 2021/09/09 at 8pm”, “Will you return?”
- “i want to travel from norwich to kent”, The station you would like to arrive at share similar names with the location you entered, here is a list a of possible stations you may be referring to: Tonbridge, Ashford International, Sevenoaks, Dartford, Tunbridge Wells. Not a bug but could say something more informative like “here is a list of stations in kent”
- “Please confirm your booking...
  Departure datetime: 2021-02-01 at 18:00
  Departing from: Norwich
  Arriving at: London Liverpool Street”. “no”, “Please respond appropriately”
- Issue with the final / next transaction. “What else can I help you with?” “Ticket” “Is there anything else I can help with”. Does not seem to recognise this. “no” works but this does not make sense for proper language.
- “What station are you travelling to?” “Forest gayt” “Did you mean Forest Gate?” “What station are you travelling to”. Same response fired and confirmation does not work

Person A1: Wants to book a single ticket step by step, making no mistakes

Expected outcome: Each response by the bot is appropriate, reaching to the goal of a train ticket

Observed outcome:

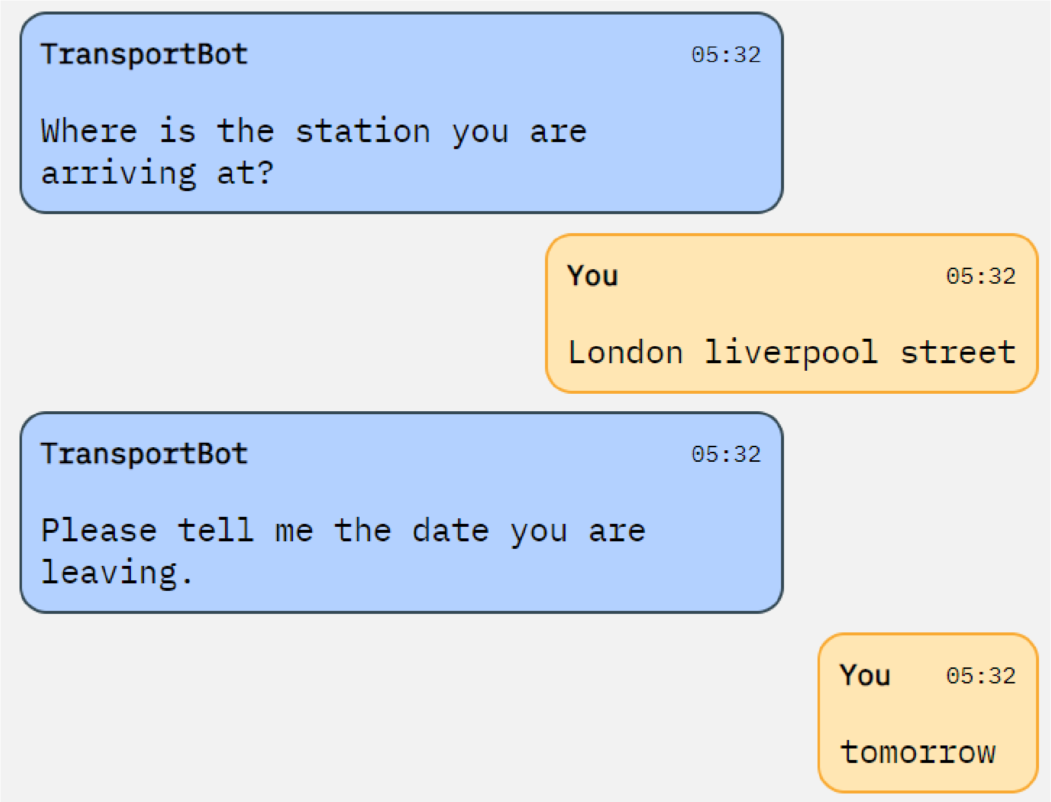


TransportBot 05:31
Hello! How may I help you today?

You 05:31
I would like to book a ticket

TransportBot 05:32
Where are you travelling from?

You 05:32
Norwich



TransportBot 05:32

Where is the station you are arriving at?

You 05:32

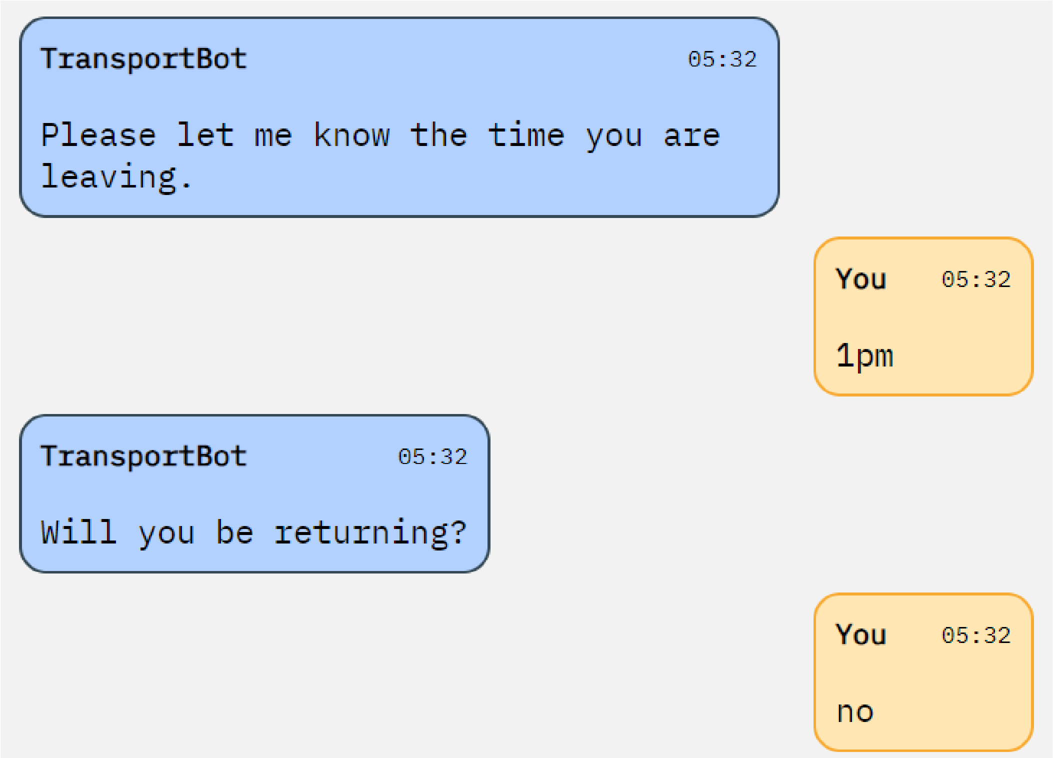
London liverpool street

TransportBot 05:32

Please tell me the date you are leaving.

You 05:32

tomorrow



TransportBot 05:32

Please let me know the time you are leaving.

You 05:32

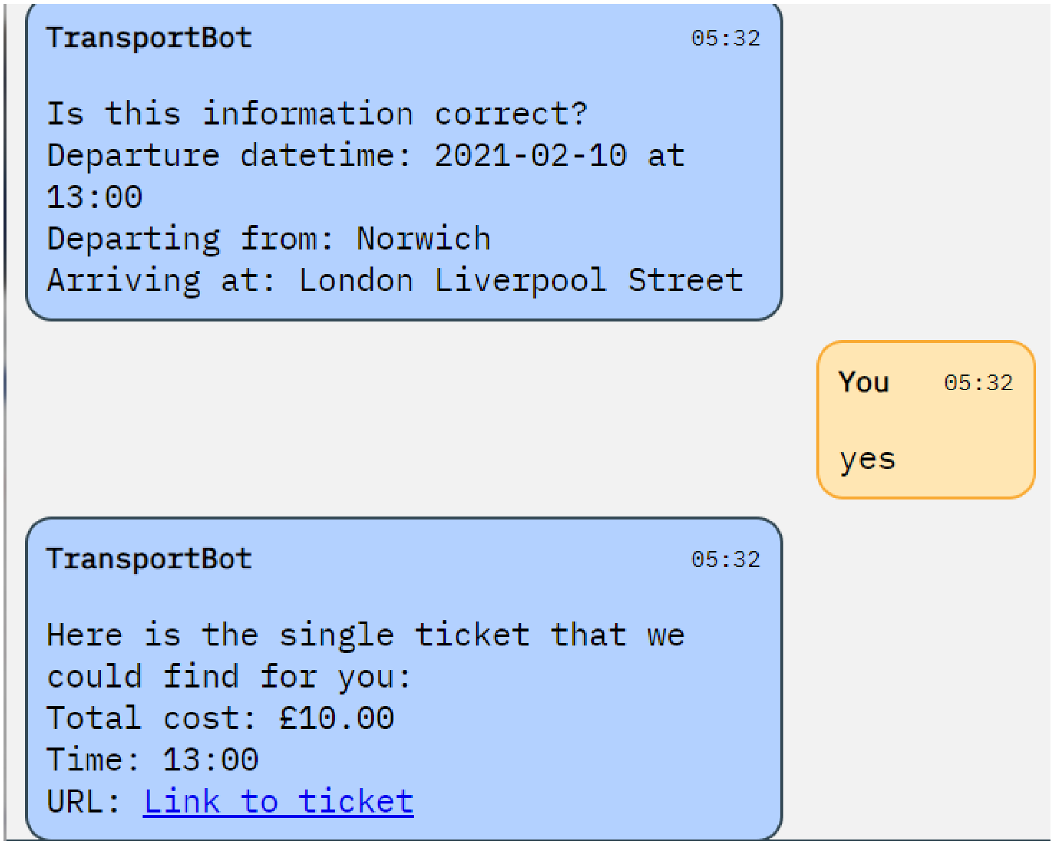
1pm

TransportBot 05:32

Will you be returning?

You 05:32

no



TransportBot 05:32

Is this information correct?
Departure datetime: 2021-02-10 at 13:00
Departing from: Norwich
Arriving at: London Liverpool Street

You 05:32

yes

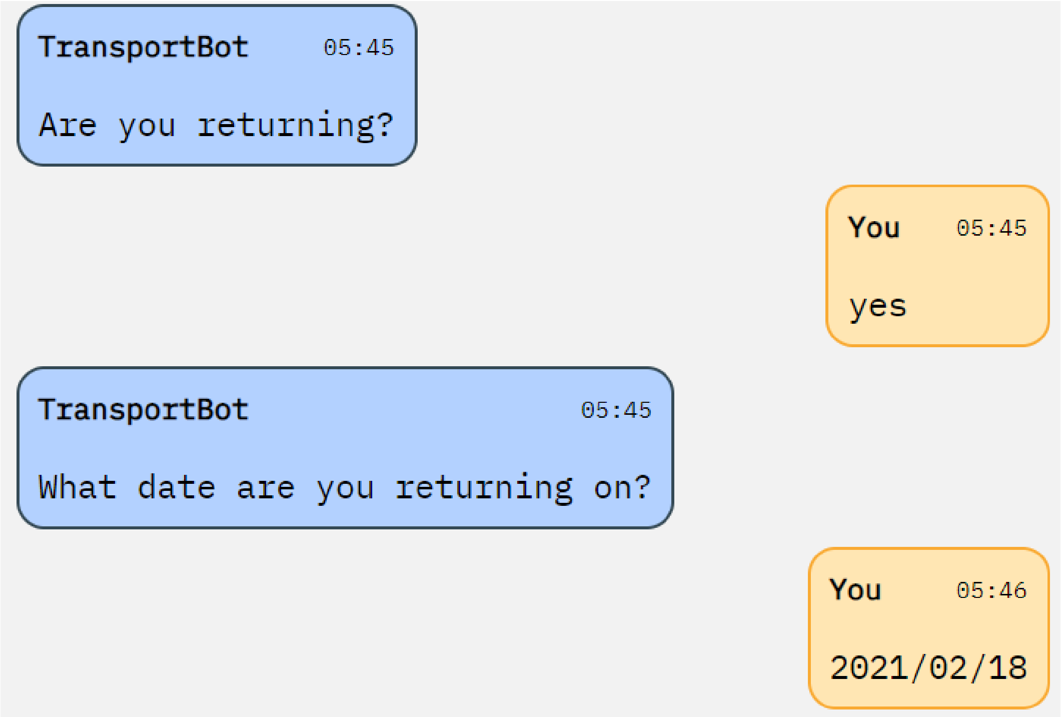
TransportBot 05:32

Here is the single ticket that we could find for you:
Total cost: £10.00
Time: 13:00
URL: Link to ticket

Person A2: Wants to book a return ticket step by step, making no mistakes

Expected outcome: Each response by the bot is appropriate, reaching to the goal of a return train ticket

Observed outcome:



TransportBot 05:45

Are you returning?

You 05:45

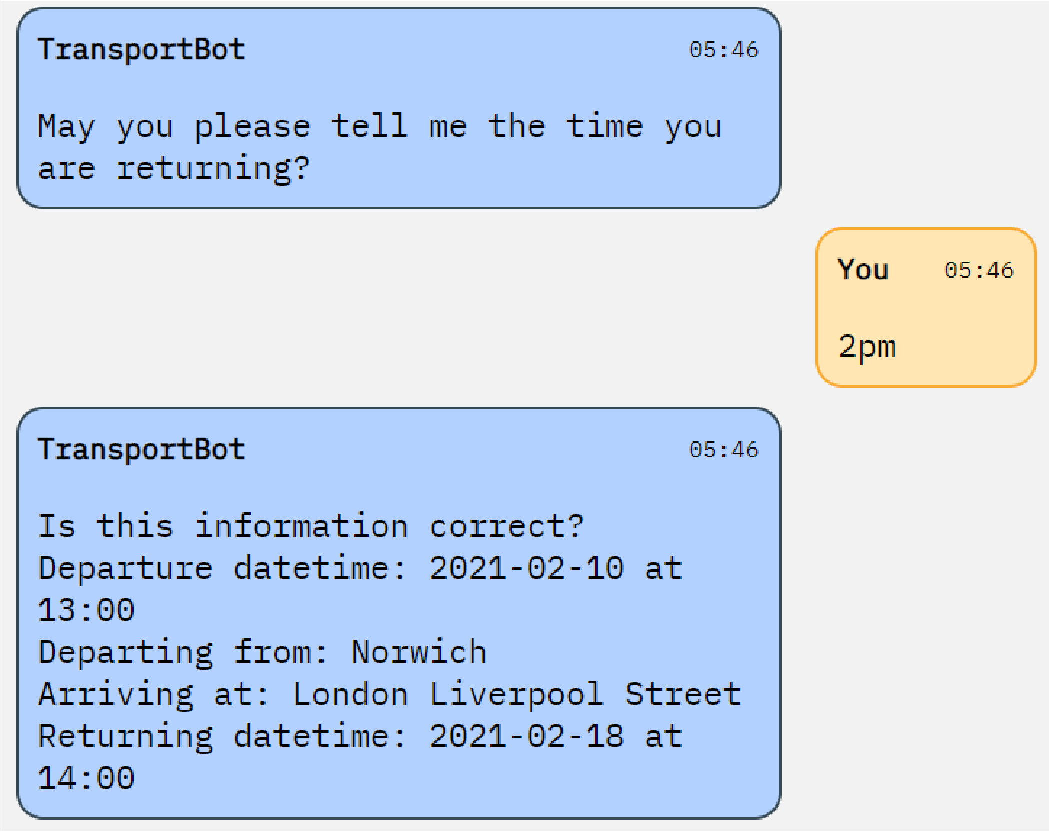
yes

TransportBot 05:45

What date are you returning on?

You 05:46

2021/02/18



TransportBot 05:46

May you please tell me the time you are returning?

You 05:46

2pm

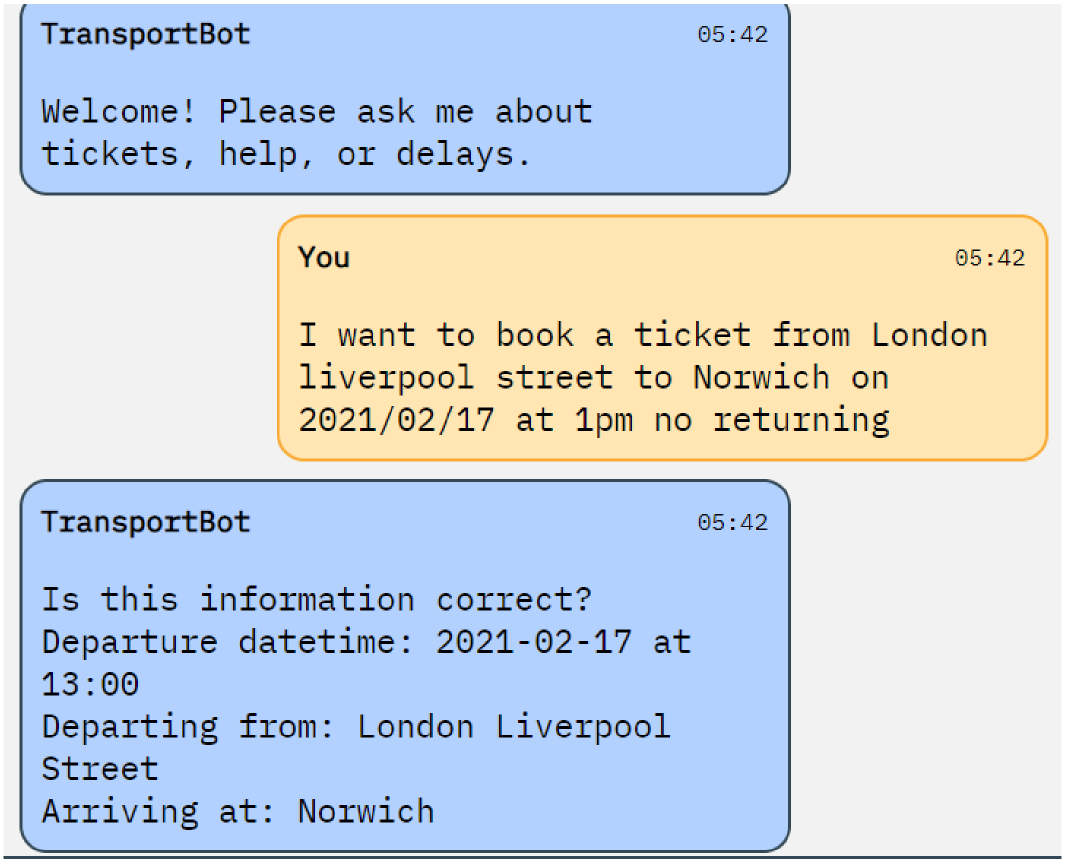
TransportBot 05:46

Is this information correct?
Departure datetime: 2021-02-10 at 13:00
Departing from: Norwich
Arriving at: London Liverpool Street
Returning datetime: 2021-02-18 at 14:00

Person B: Wants to book a single ticket in a single query

Expected outcome: The bot responding with a confirmation of all the details correctly matching what the user entered

Observed outcome:



TransportBot 05:42

Welcome! Please ask me about tickets, help, or delays.

You 05:42

I want to book a ticket from London liverpool street to Norwich on 2021/02/17 at 1pm no returning

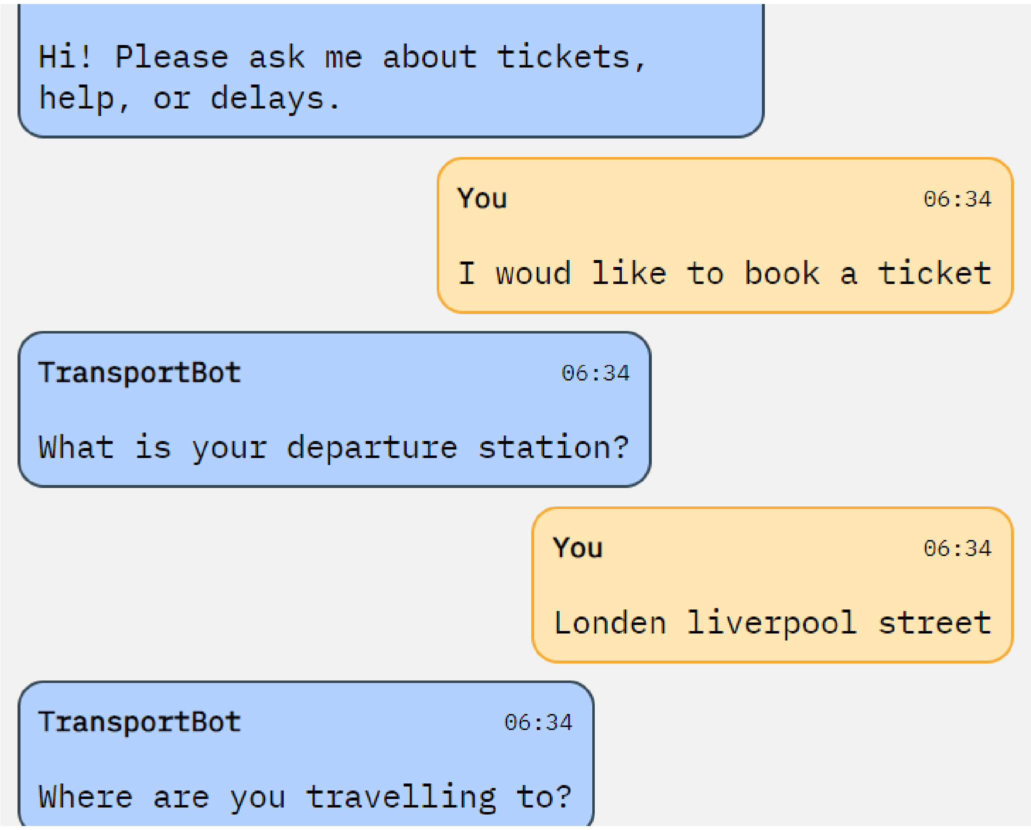
TransportBot 05:42

Is this information correct?
Departure datetime: 2021-02-17 at 13:00
Departing from: London Liverpool Street
Arriving at: Norwich

Person C: Wants to book tickets step by step but makes mistakes in typing

Expected outcome: User should be corrected/suggested alternative answers at any stage of the booking process

Observed outcome:



Hi! Please ask me about tickets, help, or delays.

You 06:34

I woud like to book a ticket

TransportBot 06:34

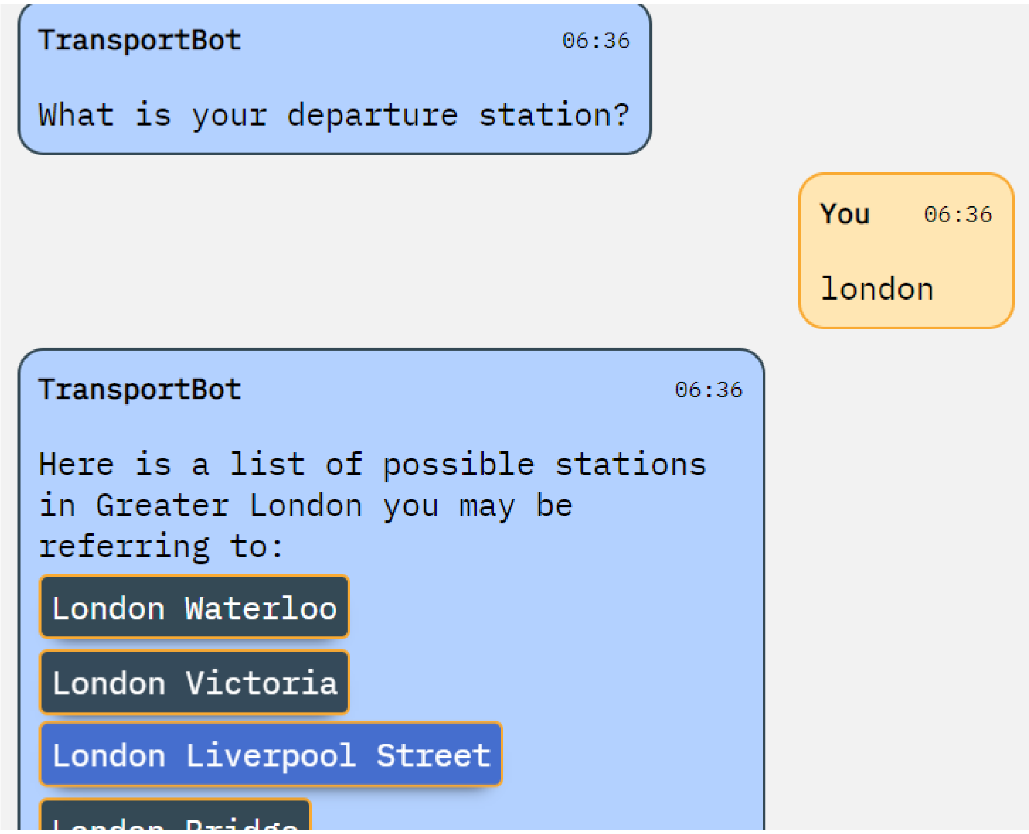
What is your departure station?

You 06:34

Londen liverpool street

TransportBot 06:34

Where are you travelling to?



TransportBot 06:36

What is your departure station?

You 06:36

london

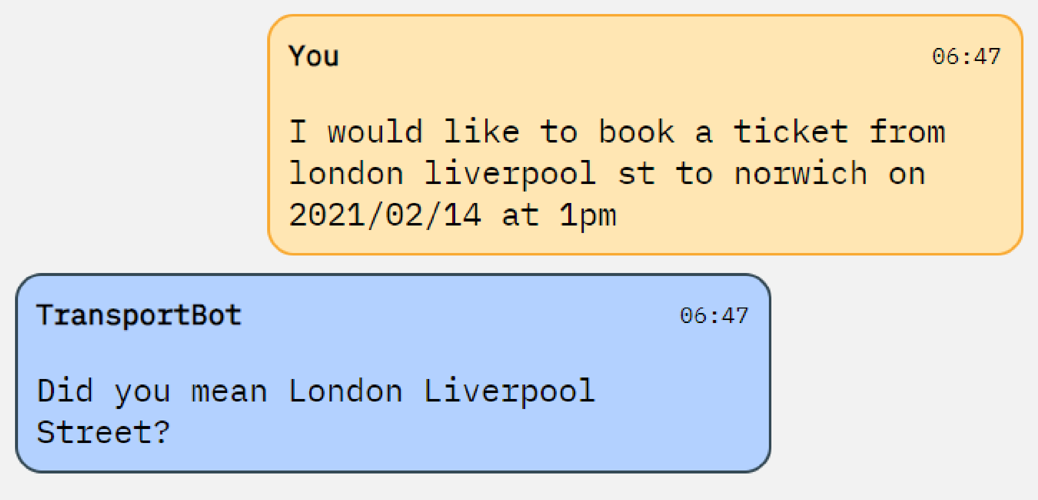
TransportBot 06:36

Here is a list of possible stations in Greater London you may be referring to:

Person D: Wants to book tickets in a single query but makes a spelling mistake

Expected outcome: The spelling mistake is caught and the user is prompted to enter the information correctly

Observed outcome:



You 06:47

I would like to book a ticket from london liverpool st to norwich on 2021/02/14 at 1pm

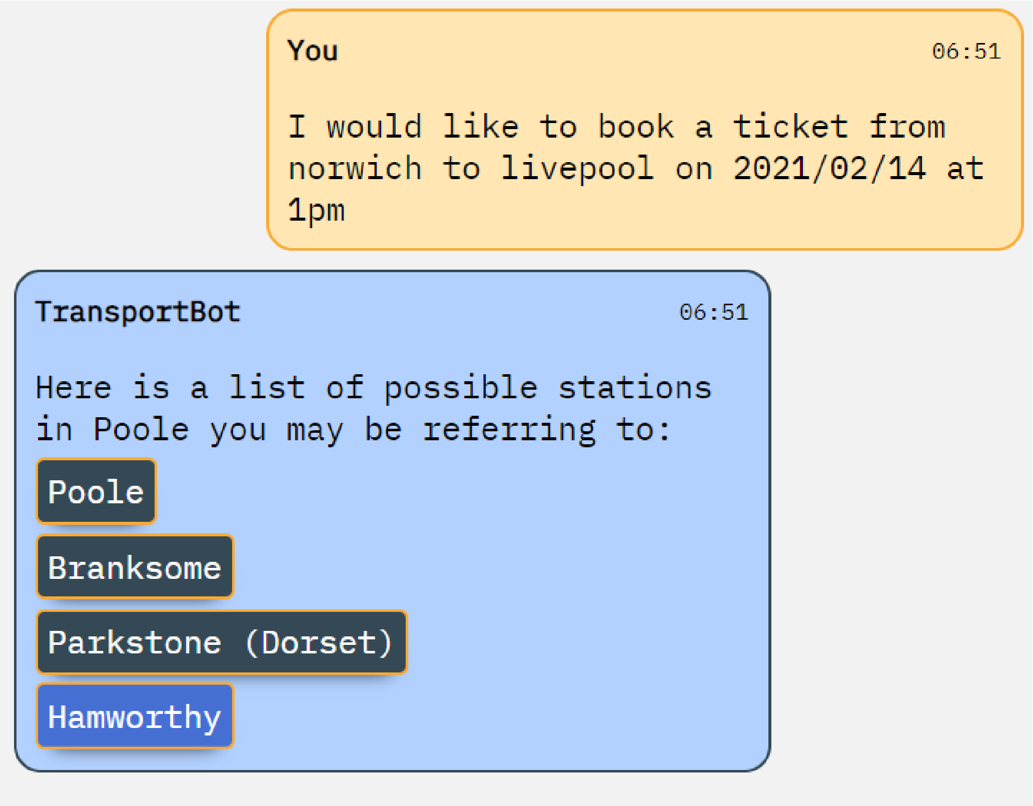
TransportBot 06:47

Did you mean London Liverpool Street?

Person E: Wants to book tickets but does not know the destination station

Expected outcome: If the user knows the location/county of the station the bot would then suggest the top 5 most popular results for that county

Observed outcome:


You 06:51

I would like to book a ticket from norwich to livepool on 2021/02/14 at 1pm

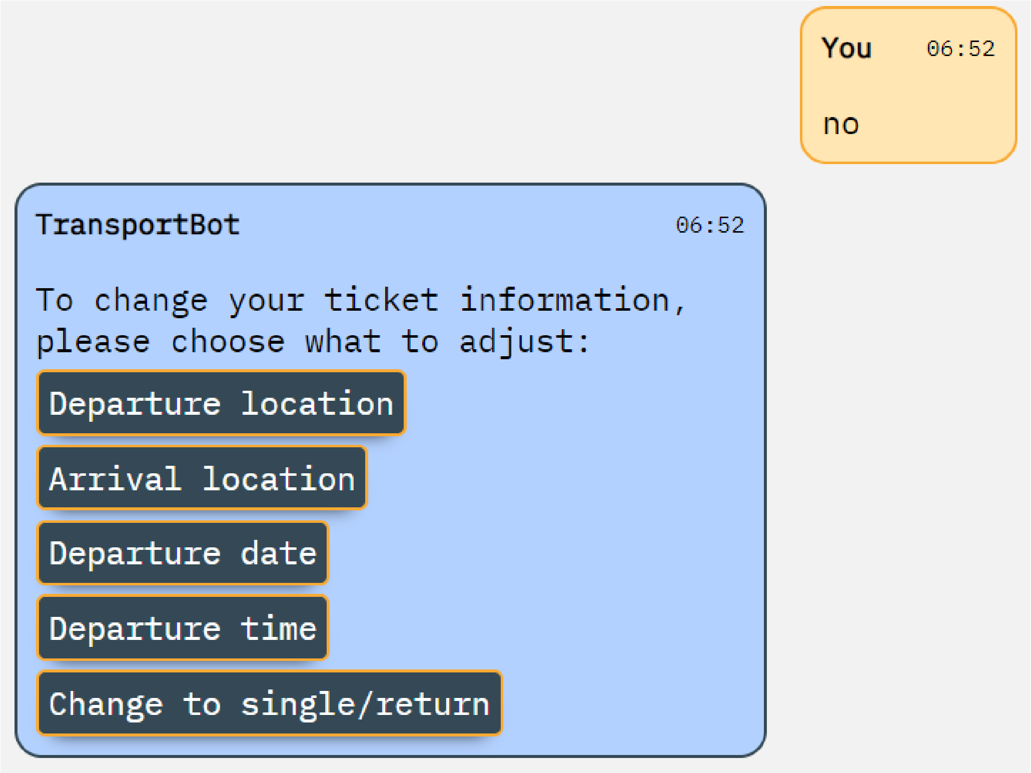
TransportBot 06:51

Here is a list of possible stations in Poole you may be referring to:

Person F: Wants to book tickets but changes their mind and needs to change their travel date / time

Expected outcome: The user can pick an option to change their ticket information

Observed outcome:


You 06:52

no

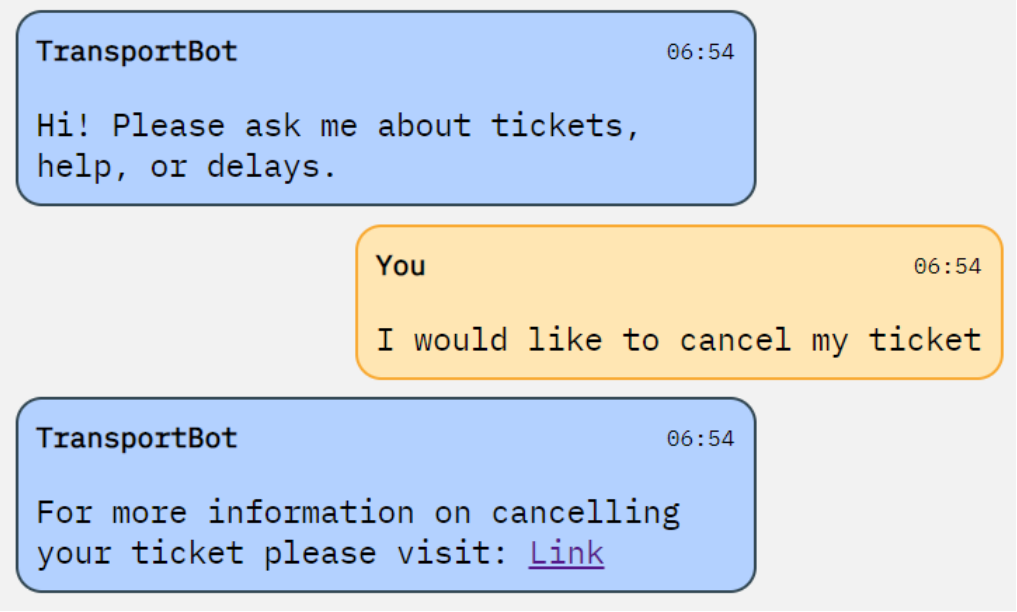
TransportBot 06:52

To change your ticket information, please choose what to adjust:

Person G: Asks cancellation info

Expected outcome: The bot would reply with a link directing them to more information about cancelling their tickets

Observed outcome:


TransportBot 06:54

Hi! Please ask me about tickets, help, or delays.

You 06:54

I would like to cancel my ticket

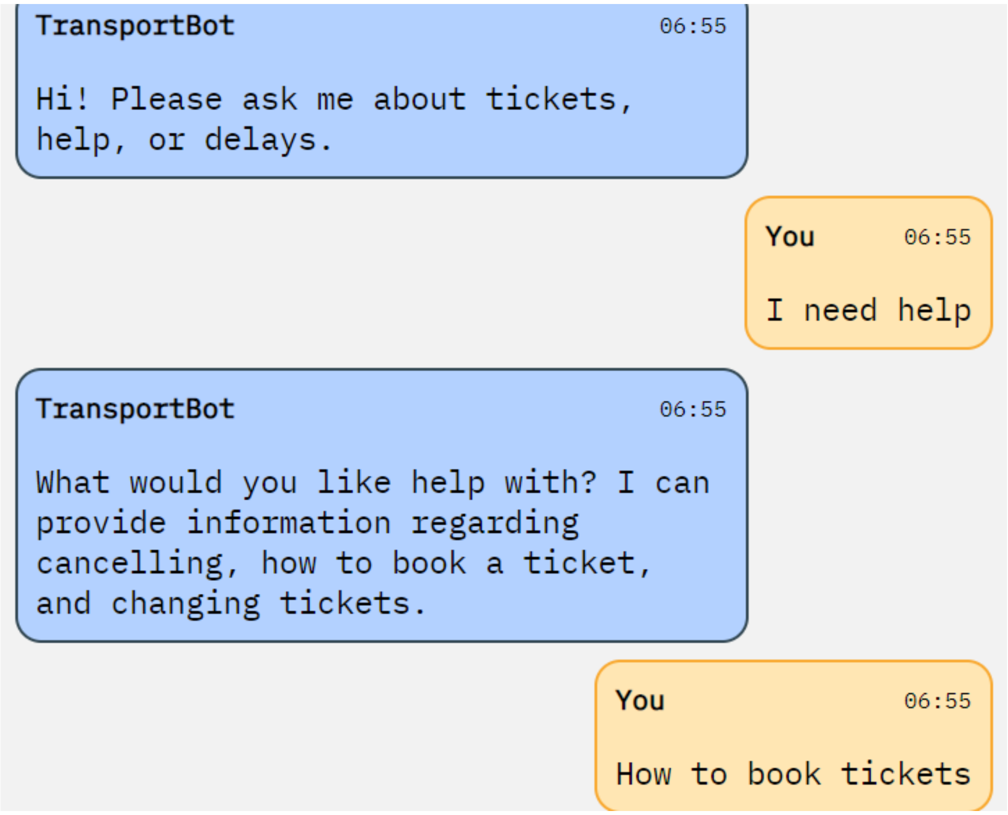
TransportBot 06:54

For more information on cancelling your ticket please visit: Link

Person H: Needs help using the bot

Expected outcome: The bot would offer advice on how to book tickets and give an example the user requires further assistance

Observed outcome:



TransportBot 06:55

Hi! Please ask me about tickets, help, or delays.

You 06:55

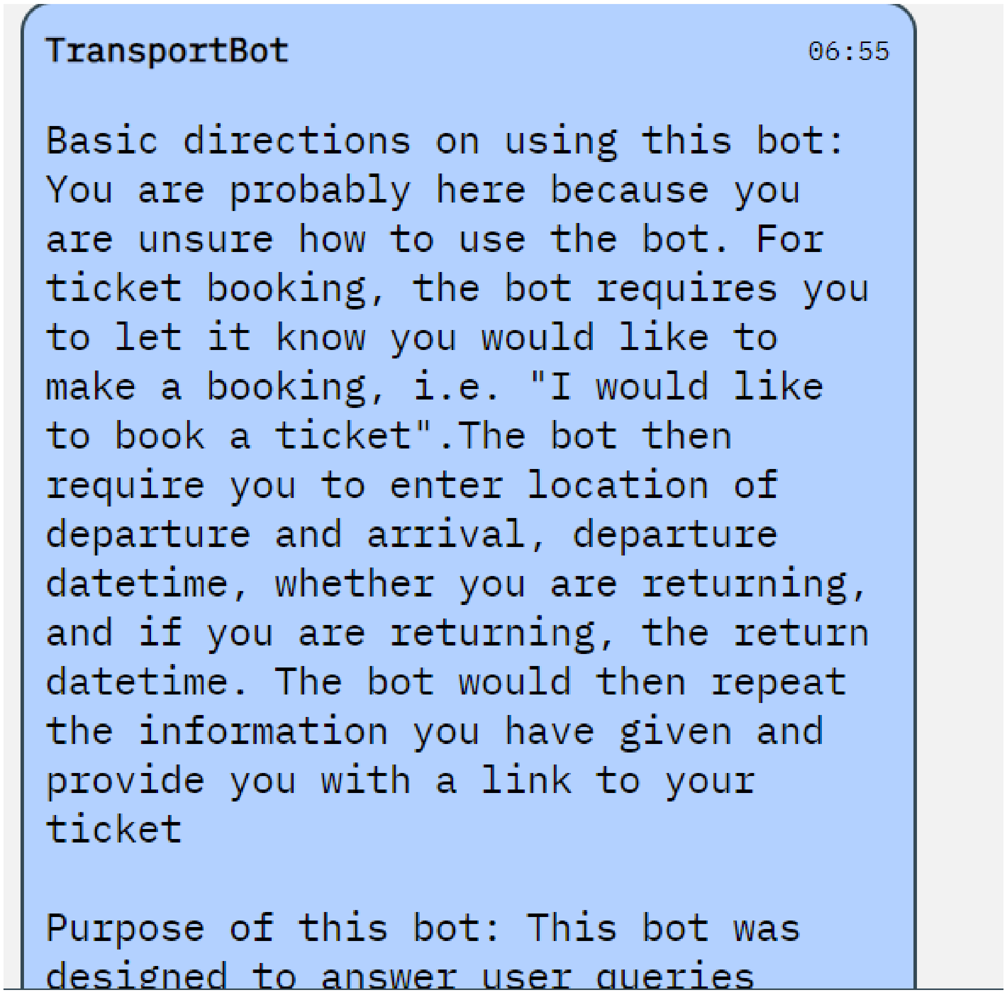
I need help

TransportBot 06:55

What would you like help with? I can provide information regarding cancelling, how to book a ticket, and changing tickets.

You 06:55

How to book tickets



TransportBot 06:55

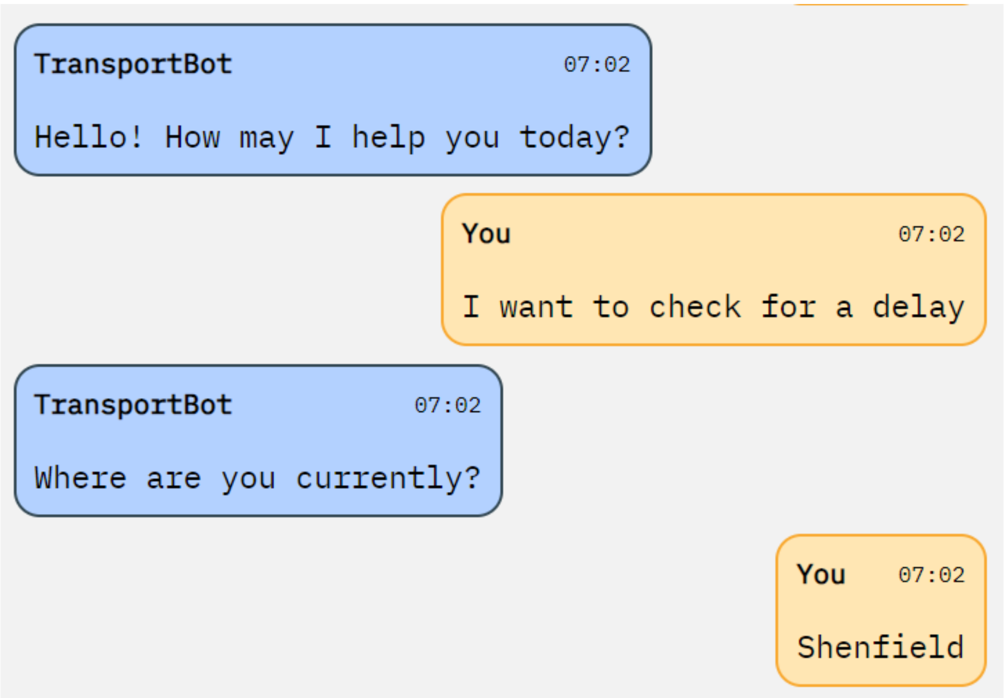
Basic directions on using this bot: You are probably here because you are unsure how to use the bot. For ticket booking, the bot requires you to let it know you would like to make a booking, i.e. "I would like to book a ticket".The bot then require you to enter location of departure and arrival, departure datetime, whether you are returning, and if you are returning, the return datetime. The bot would then repeat the information you have given and provide you with a link to your ticket

Purpose of this bot: This bot was designed to answer user queries

Person I: Want to check delays on their journey step by step

Expected outcome: The user can either enter the

Observed outcome:



TransportBot 07:02

Hello! How may I help you today?

You 07:02

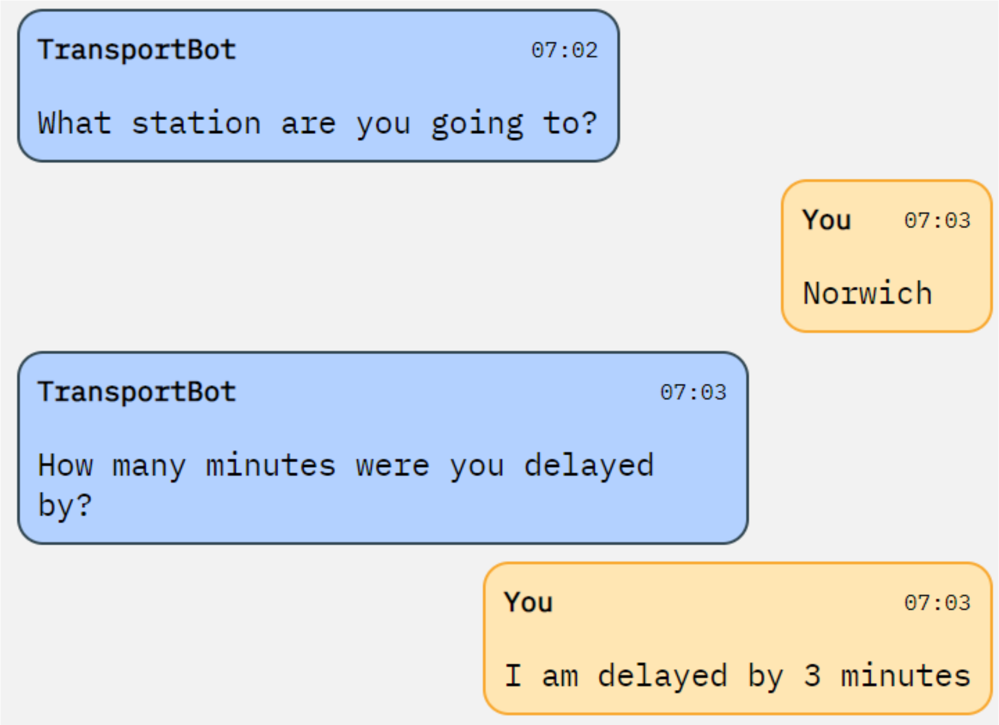
I want to check for a delay

TransportBot 07:02

Where are you currently?

You 07:02

Shenfield



TransportBot 07:02

What station are you going to?

You 07:03

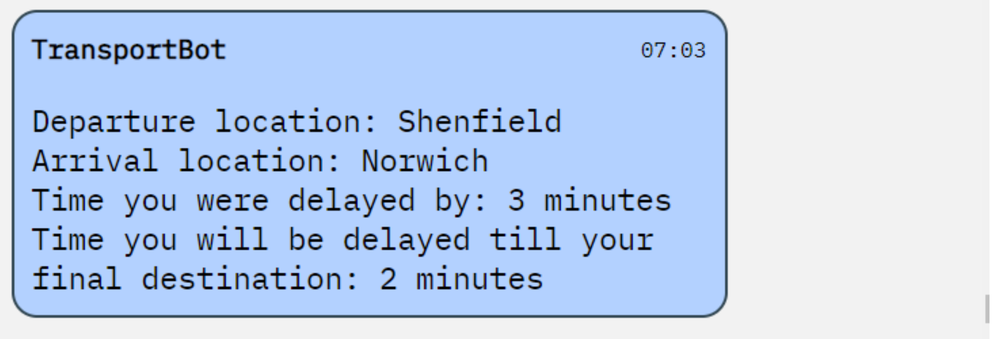
Norwich

TransportBot 07:03

How many minutes were you delayed by?

You 07:03

I am delayed by 3 minutes

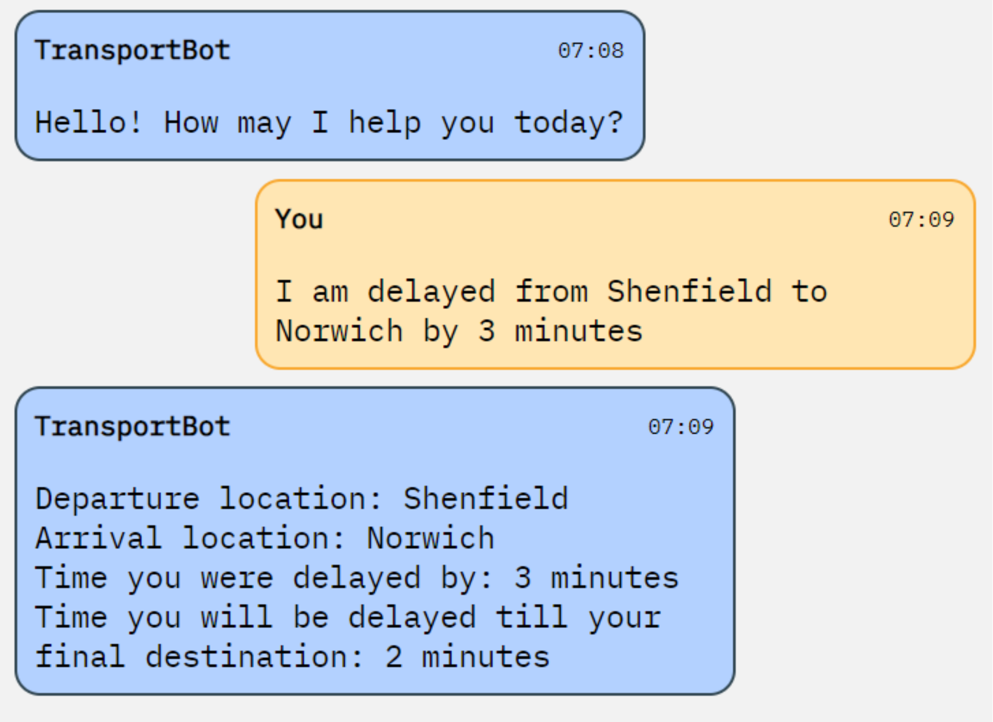

TransportBot 07:03

Departure location: Shenfield
Arrival location: Norwich
Time you were delayed by: 3 minutes
Time you will be delayed till your final destination: 2 minutes

Person J: Want to check delays on their journey in a single query

Expected outcome: The user can either enter the

Observed outcome:


TransportBot 07:08

Hello! How may I help you today?

You 07:09

I am delayed from Shenfield to Norwich by 3 minutes

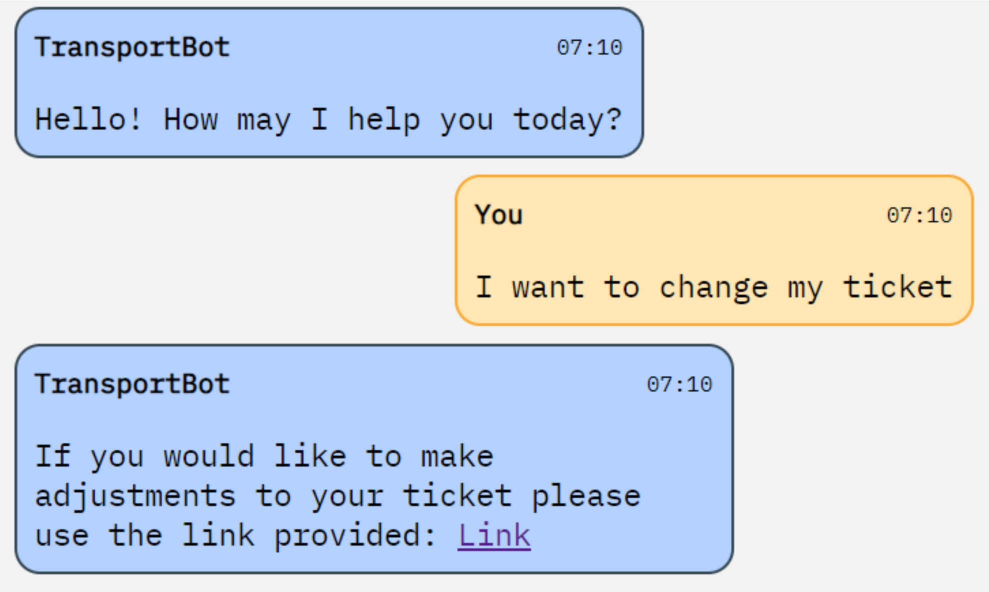
TransportBot 07:09

Departure location: Shenfield
Arrival location: Norwich
Time you were delayed by: 3 minutes
Time you will be delayed till your final destination: 2 minutes

Person K: Asks for help regarding changing tickets

Expected outcome: The user would be directed to a link with further information about changing their purchased ticket

Observed outcome:


TransportBot 07:10

Hello! How may I help you today?

You 07:10

I want to change my ticket

TransportBot 07:10

If you would like to make adjustments to your ticket please use the link provided: Link

Evaluation

Due to the lack of experience using PyKnow, the approach taken in developing the engine has certain drawbacks. We believe that the rules for the questions and answers could have been reworked in a different style thus leaving the piece of code in a more maintainable fashion for individuals who would want to understand it or even make use of it. Currently, the engine receives a dictionary of the processed input from the user, that information passes through the corresponding query, however, that same information is also passed onto the next query if validation is successful. A change can be made; however, a huge overhaul of the reasoning engine must be done for it to be functioning at the same level as what we have currently. A solution to this inconvenience in the code can be splitting the query into two separate rules to prevent the previous information trickling into the next set of queries.

The reasoning engine may have also been developed earlier if we knew how much the NLP is closely interlinked with the reasoning engine. Most of the critical components for the engine were only developed after the NLP was ready to be used, thus implying that if we had figured earlier what major components of the chat bot should have higher priority in development then the reasoning engine could have possibly spent more time in being finer tuned towards answering train ticket/delay queries.

Delay Prediction Models

Process

The first decision we made was to create separate models for each station as opposed to one single model for the entire network. Although this is a little more complex than an entire route model, it would mean the system would be easily expandable in the future should more stations be tested, without affecting the accuracy of previously modelled stations. We decided to start with a designing a KNN model as it is relatively simple to test, we chose to do this testing in a Jupyter notebook as this makes it easier to change and keep track of values and make quick changes to the model as we go.

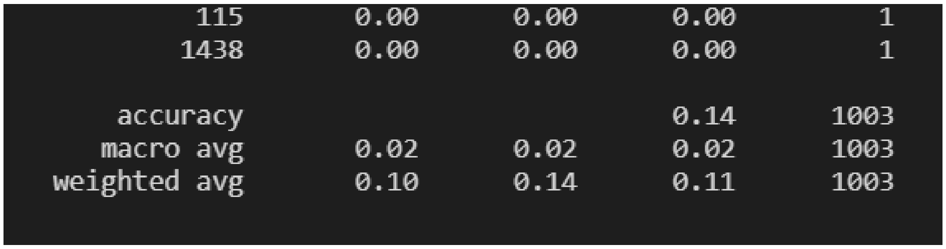
Our initial dataset was all the rdg data provided through nrch-livst-test-data, plus all the a51 data from August 2018 to March 2019. we processed this data to remove much of it and encode categorical columns to numerical ones. The details we selected for this test were the day of the week (1-7), weekday (0, 1), off-peak (0, 1), hour of day (0-23) and delay (approximately -10 to 60).

KNN Model

For our first attempt, we wanted to use a classification algorithm and decided on KNN as it is simple to test, being a form of lazy learning algorithm, and our dataset has relatively low dimensionality. Firstly, we decided to see how accurately we could find the delay time for trains coming into Stratford (STFD) and heading for Bethnal Green (BTHNLGR). We loaded our dataframe with 10027 matching entries from the database.

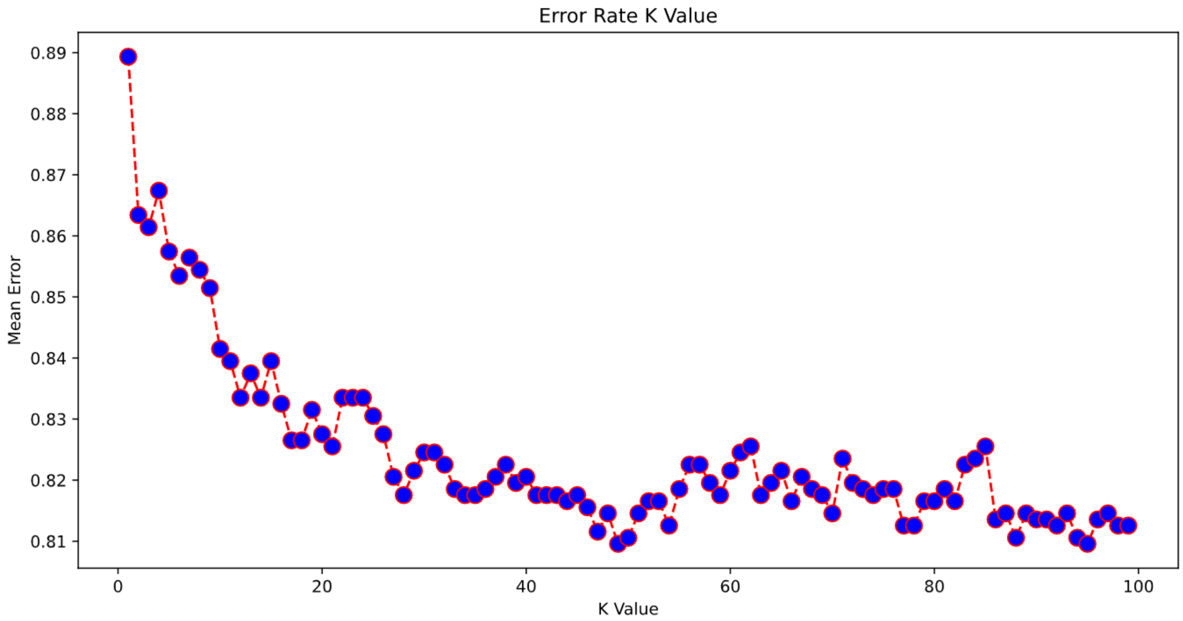
We selected delay as the label (y) and the remaining numerical columns as our attributes, then split the dataset 90/10 (training, testing). We chose those attributes previously since they were easily obtainable from the data we had collected, and the split was chosen due to feeling the more training data we have, the more accurate the model may be. Since there is no way to train the model based off of the testing data, we wanted to ensure the model had as much data as possible to work with.

We scaled the attributes using sklearn’s StandardScaler preprocessor to ensure each attribute has a proportional contribution to the final distance, applied the KNN classifier with a starting K value of 5 and made the predictions. For evaluation, we created a confusion matrix and classification report, the totals of which you can find below.



```
         115       0.00      0.00      0.00         1
        1438       0.00      0.00      0.00         1

    accuracy                           0.14      1003
   macro avg       0.02      0.02      0.02      1003
weighted avg       0.10      0.14      0.11      1003
```

An accuracy of 0.14 is quite poor, it tells us that only 14% of the test data was correctly predicted by the model. Although we do not expect a huge amount of accuracy in the results, it feels like random sampling would get just as close as this. we decided to see if changing the value of K between 1 and 100 would have any effect on this result and plotted the results to a graph as shown below.



From this we can tell that the best K value for our dataset is probably around 50, but that still only gives us an accuracy of around 19%. Although 19% is quite a bit more accurate than the previous 14% (a gain of around 35%), we feel we can make further improvements.

Evaluation of the KNN Model

The model seems to work, but not very well. We feel this is down to the data set being flawed more than the model, but we will be adjusting both. We will reset K to 10 as default, since that should be a good starting point with this volume of data but will again try to find the best K value using the same technique used on KNN, on KNN2. We realised from looking at our test data that entries for London Liverpool Street were accidently left out due to a programming error in what data should be loaded from CSV, so we will change the data loading system to include these. we will also be adding a new attribute, delay_change, which will be the difference between the previous delay value and the new delay. We will then adjust the model to use delay_change as our label and delay will become another attribute. We are testing this change since we feel if we compare the current delay time with the other factors, we may be able to find a pattern in the change of delay e.g., do all trains that came into station X 5 minutes late gain the same amount of time between arriving at station X and arriving at stop Y?

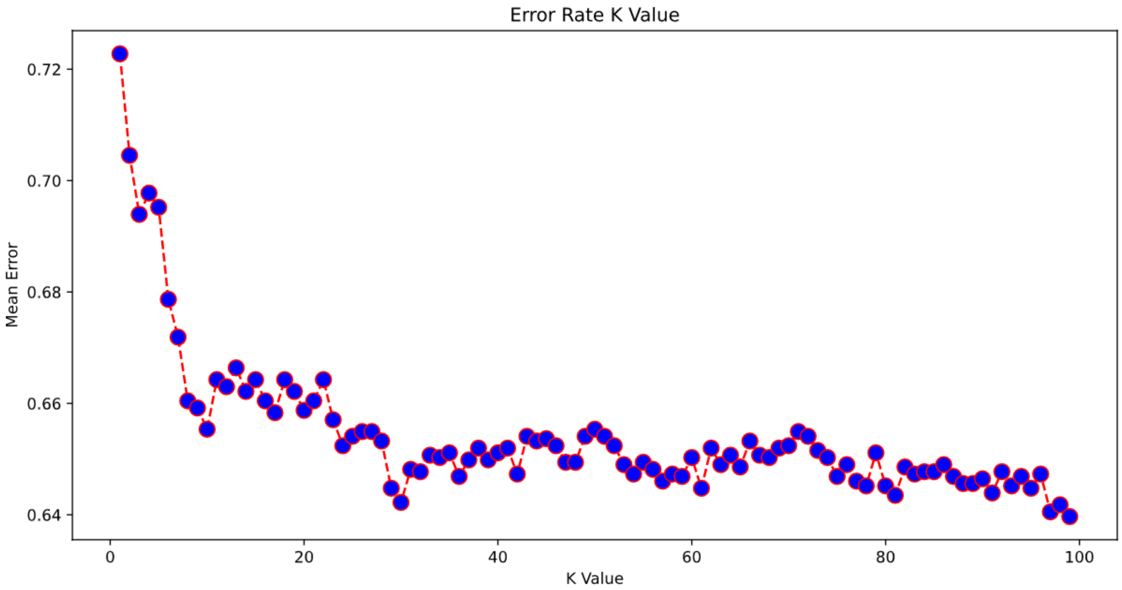
KNN Model 2

After making the changes to the data and model noted above, we re-ran the KNN model, using a 90/10 training/testing split and a K value of 10. The following table shows the results given from the classification report:

value	**precision**	**recall**	**f1-score**	**support**
-1440	0.00	0.00	0.00	2
-1439	0.12	0.33	0.18	3
-1438	0.00	0.00	0.00	2
-1436	0.00	0.00	0.00	1
-1435	0.00	0.00	0.00	1
-1433	0.00	0.00	0.00	1

-17	0.00	0.00	0.00	1
-7	0.00	0.00	0.00	1
-5	0.00	0.00	0.00	1
-4	0.00	0.00	0.00	2
-3	0.00	0.00	0.00	2
-2	0.00	0.00	0.00	11
-1	0.16	0.06	0.09	136
0	0.41	0.63	0.50	837
1	0.31	0.35	0.33	582
2	0.23	0.14	0.17	407
3	0.20	0.07	0.11	191
4	0.10	0.03	0.05	98
5	0.05	0.02	0.03	47
6	0.00	0.00	0.00	14
7	0.00	0.00	0.00	8
8	0.00	0.00	0.00	2
10	0.00	0.00	0.00	1
14	0.00	0.00	0.00	1
16	0.00	0.00	0.00	1
18	0.00	0.00	0.00	1
24	0.00	0.00	0.00	1
27	0.00	0.00	0.00	1
28	0.00	0.00	0.00	1
32	0.00	0.00	0.00	1
1439	0.00	0.00	0.00	1
accuracy			0.34	2359
macro avg	0.05	0.05	0.05	2359
weighted avg	0.29	0.34	0.30	2359

An accuracy of 34% is a notable improvement, even without adjusting the K value, however we are now concerned about the unusual values being generated outside of -17 to 32. These indicate that there was probably an issue with trains travelling from one day to another, as 1440 minutes is 24 hours, and most of these extreme values are close to this. Since these outliers only account for 11 values in 2359 (around half a percent), it is not a major issue at this point but will be investigated a little later. Focusing on the K value for now, we conduct a similar test to before, testing all K values between 1 and 100, measuring the accuracy of the output and mapping the results to a graph, which gave the following result.



We can see that with a K value of 30, the mean error reduces to approximately 0.65, which is reasonable considering that we are attempting to predict future events based off collected data. Though it should be noted that for a K value between 11 and 100, the error only varies by 0.03 at most, with only a slight trend towards error reduction.

Removing all zero values for delay_change in our data set results in an accuracy change of less than one percent. We found that changing the distance metric from Euclidian to Manhattan gives us a similar pattern when testing values of K between 1 and 100, though it appears that across the board the mean error drops by approximately 0.01. Whether this is an artifact from this dataset or not remains to be seen, but for now we will continue to use the Manhattan Distance metric. Interestingly, this also turned the R-squared value from negative (as it was in the previous two tests) to positive, giving a value of 0.4988. This means that almost 50% of the variability in the delay change variable has been accounted for in the model, which sounds very reasonable.

We decided to check another couple stations on the network to see if the prediction would have a similar accuracy. For our first test, we chose Ipswich to Manningtree since it is further up the line, where we imagine there will not be quite as much variance in the delays. This gave a much greater change in the results, with an accuracy of 0.49 – almost half correct guesses! Since there are no trains that pass-through Ipswich into London after 23:59, we did not have as many outliers this time, though there were 4 trains that had the maximum delay increase of 30 minutes between the previous stop and Ipswich that were never correctly guessed, which makes sense since it would be almost impossible to guess if a particular train would break down between two specific stops – if our model could predict that, we'd be very rich indeed.

The second journey to check on this model was Diss to Stowmarket, which is very early on in the journey. We expected to get quite a high accuracy for this stop since it seemed it would be quite rare that the trains would be running late so early into the journey, and we were correct. We had an accuracy of 0.70, though there were only three differing values of delay_change, -1, 0 or 1 minutes, and the majority of those were 0 indicating no difference in the delay.

Evaluation of KNN Model 2

This model performed much better than the first due to changing the label (y value) and making changes to the underlying data set. We did find that in our tests of trains at Stratford, there were large outliers due to the day changing – this is a bug that will need to be fixed in the dataset before revaluation. Overall, we are happy with the performance of the KNN2 model and have decided to add it to the main project to test its accuracy on a set of journeys and see how reasonable the answers it gives are when presented with a real-world scenario as presented by the user.

There appear to be further anomalies with the dataset such as with RID code 201702077101431, which give large delay times due to miscalculation by DARWIN. To combat this, we will limit the delay to 30 minutes when parsing the data. In addition, our system requires modification to allow for accepting data from trains travelling from London to Norwich as well as further data to complete 2019 and data on Sunday trains, which requires scraping and additional data acquisition systems to convert HSP data to the DARWIN format.

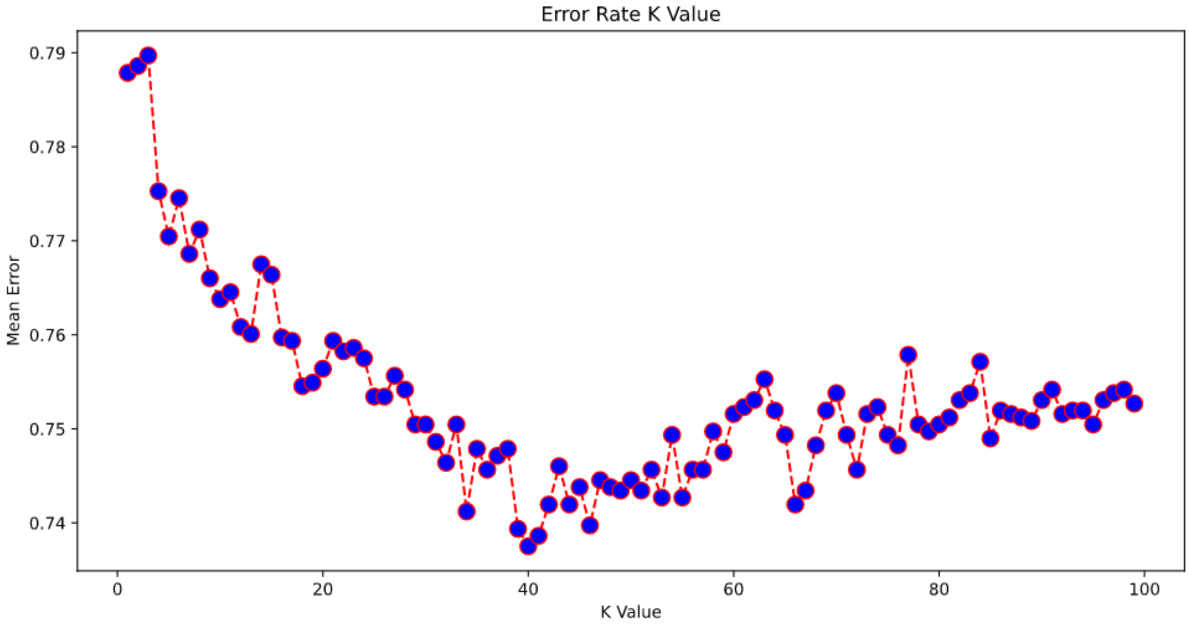
KNN Model 3

Following the analysis of KNN2 we added the further data to the model – 9 months of weekday + Saturday trains plus three years of Sunday trains - to see if this had any effect on the results. We also removed any non-station data from the given (nrch-livst-test-data) dataset and re-ran our data loading scripts so that any data that used to point to a passing point (such as Bethnal Green) now pointed to a station (in this case, London Liverpool Street). It is possible we lost a small amount of accuracy in our model though this change, but it was necessary to accurately add the scraped HSP data to our dataset.

A comparison of the KNN2 and KNN3 models are listed on the below table. Sadly, we did not take as much information as we would like from KNN2 before changing the dataset, so we only have the accuracy score to go off of. To compute the new R-squared, MSE and RMSE values, we ran the model on the dataset one hundred times splitting it differently each time to ensure a fair average.

Dataset	Source Station	Destination Station	Accuracy	Average R-squared	Average MSE	Average RMSE
KNN2	Diss	Stowmarket	0.7043	-	-	-
KNN3	Diss	Stowmarket	0.6108	0.0893	4.7686	2.1837
KNN2	Ipswich	Manningtree	0.4931	-	-	-
KNN3	Ipswich	Manningtree	0.4120	0.0625	5.7086	2.3893
KNN2	Stratford	Bow Junction	0.3394	0.4988	-	-
KNN3	Stratford	Liverpool St.	0.2377	-0.0762	12.1284	3.4826
KNN3	Stratford	Shenfield	0.5221	0.1838	7.5487	2.7475
KNN3	Manningtree	Ipswich	0.4547	-0.0323	6.9748	2.6410
KNN3	Diss	Norwich	0.5029	0.0171	3.8696	1.9671

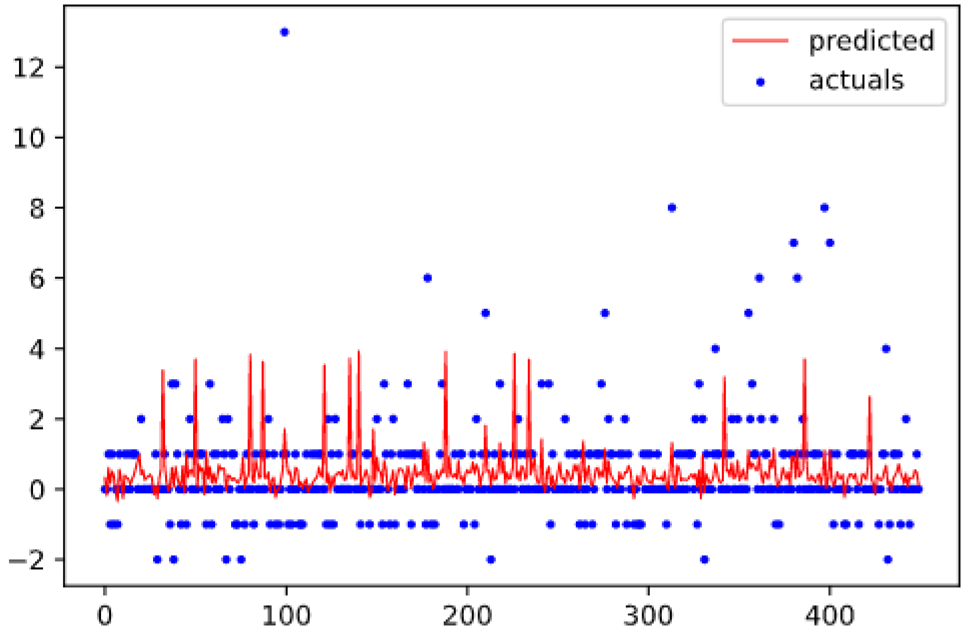
Bizarrely, with more data points our accuracy dropped on all three tests going towards London to far below where they were with the KNN2 model. What was also strange was that the trains modelled going towards Norwich had predictions that remained relatively stable across the network. We decided to evaluate the K-value again on the test from Stratford to Liverpool Street to see if having a larger K-value would help give greater accuracy.



The best K value is now at 40, but there is a lot more variation in the values compared to results in KNN2 and moving to K=40 still only brings accuracy to 0.26, far short of the 0.34 previously noted. We considered that there may be an error with our import of data from HSP, but after checking multiple times, the data imported into the database appears correct and matches what was stored in the scraped CSV files. This difference in results has truly confused us, but we decided to continue testing other models in the hope that they would work better with the new data.

Bayesian Ridge Regression Model

We wanted to use a regression model second, to see if it gave a better result. Since regression gives an average of its neighbors rather than a specific value, it is likely this model will not be useful in this case, but we wished to test this. We ran the simulation again and created a plot from 450 entries of trains between Diss and Stowmarket, which can be seen in the below chart.



We can clearly see from this that some inputs will result in drastic jumps in the change of delay, which may not be ideal for our scenario. We can also see that the prediction very rarely gives a reduction in delay. This is almost certainly due to the data not being narrowed down to just delayed trains, meaning the bulk of the data is assuming the train is coming into the station on-time. As mentioned before, we could select for only delayed trains, but this will then cause issues if our delay reduced to 0 minutes – there would be no data for the model to work with. We compared the results of our tests to the same tests performed on the KNN3 model above, once again running each model one hundred times per test for a fair average.

Dataset	Source Station	Destination Station	Accuracy	Average R-squared	Average MSE	Average RMSE
KNN3	Diss	Stowmarket	0.6108	0.0893	4.7686	2.1837
BRR	Diss	Stowmarket	-	0.0811	4.8051	2.1921
KNN3	Ipswich	Manningtree	0.4120	0.0625	5.7086	2.3893
BRR	Ipswich	Manningtree	-	0.1078	5.6067	2.3678
KNN3	Stratford	Liverpool St.	0.2377	-0.0762	12.1284	3.4826
BRR	Stratford	Liverpool St.	-	0.1705	9.5594	3.0918
KNN3	Stratford	Shenfield	0.5221	0.1838	7.5487	2.7475
BRR	Stratford	Shenfield	-	0.0611	8.2939	2.8799
KNN3	Manningtree	Ipswich	0.4547	-0.0323	6.9748	2.6410
BRR	Manningtree	Ipswich	-	0.0333	6.5167	2.5528
KNN3	Diss	Norwich	0.5029	0.0171	3.8696	1.9671
BRR	Diss	Norwich	-	0.0394	3.8515	1.9625

Since it is not possible to get an accuracy score with a regression model, we must use the R-Squared and MSE/RMSE scores instead. We can see that this model performs very slightly better in most scenarios compared to KNN3, though the R-squared average is now increasing for trains heading towards London as the MSE increases, which means the variation is decreasing as the accuracy decreases which was not expected. Since regression is not categorical and gives us a floating-point value as an answer, we would need to interpret the value coming out the model as a categorical value. We could also see from our tests that the larger the delay we put in, the larger the prediction value would be, which infers that the more delayed a train is, the greater the delay will be after this stop, with no prior delay showing a result around 0.83. This shows that we will need to write an algorithm to smartly interpret the numbers coming out of the model to map them to an integer value, which will likely ruin the apparent gain in accuracy from using such a model.

Evaluation of the Bayesian Ridge Regression Model

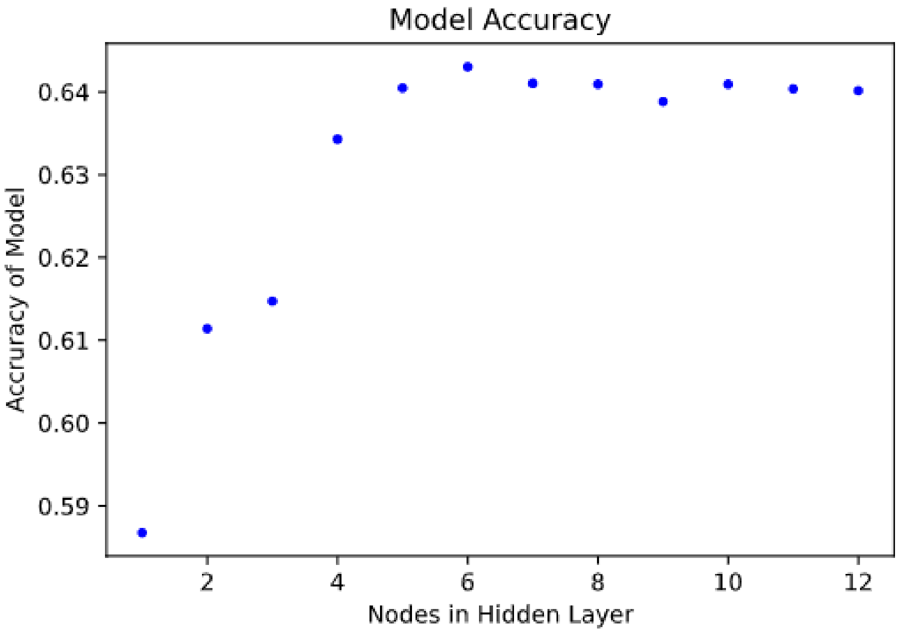
As mentioned earlier, we had a possible issue with trains that were delayed for a time different from the current train was, which could be throwing out our estimates. We would try selecting different ranges of current delay from the database when building any future BRR model to see how that affects predictions. This would start to cause issues with storing models, of course, but we have decided since the models are taking up a lot of database space (we are already storing over 120MB of data, including 50MB of KNN models) and each model takes only a fraction of a second to build with the data we have, we will not be storing and retrieving future models, but building them on-the-fly to evaluate each individual journey between two adjacent stations.

After through but undocumented experimentation, we found that the complexity of building an accurate algorithm to interpret the floating-point result from this model as a specific value makes its use in this project unlikely, as time could better be spent looking at other models with may give a similar increase in accuracy, so we will leave our research on Bayesian Ridge here.

Artificial Neural Network Model

We wanted to try our hand at classifying via a Multi-layer Perceptron Classifier for our third model to see if a more complex model will result in more accurate results. Since this ANN is categorical, we will not have the issue faced with the Bayesian Ridge model as we can classify the change in delay as a single integer instead. We feel that outlier values may give us issues however, but this is something we will investigate later. For now, we decided on using 30% of the dataset for testing, as we can use backpropagation to ensure the model still learns from testing. We are using the Hyperbolic Tan (tanh) activation function and the default Adaptive Moment Estimation (Adam) optimisation algorithm across a maximum of five hundred epochs for our first try.

We started by using the Diss to Stowmarket subset of data, testing a range of neurons in a single hidden layer from 1 to 10 to find the highest accuracy. These results are shown in the graph below.



From this graph we can see the accuracy of the model levels out at around five or six nodes at approximately 64%. We compared the R-squared and MSE of these two models and found that five nodes had the more favorable error values, with an R-squared of 0.0716 and MSE of 4.8473 compared to 0.0551 and 4.9332 from six nodes. These results place the model around the same place as our previously tested models, slightly better in accuracy compared to KNN3 but with a lower R-score.

A test using the logistic sigmoid activation function did not give any better results, with an accuracy of 64% again, R-squared values of 0.0642 and 0.0499 for five and six nodes respectively, and MSEs of 4.8856 and 4.9604. The models required a larger number of iterations to train however, so we decided to stick with tanh for now. We also tried changing the solver from the default Adam solver to a stochastic gradient descent (sgd) which resulted in an accuracy of only 62% at best, tests using the limited-memory BFGS (lbfgs) solver gave us similar results. Therefore, we will stick with the Adam solver from now on.

Next, we wanted to see if adding a second hidden layer would give us better results. We decided to stick with testing five and six nodes on layer one and then tested between four and seven nodes on layer two. The results of these tests are shown below.

Hidden Layer 1	Hidden Layer 2	Accuracy	R-squared	MSE
5	4	0.6385	0.0515	4.9523
5	5	0.6421	0.0916	4.7430
5	6	0.6399	0.0624	4.8952
5	7	0.6409	0.0660	4.8767
6	4	0.6419	0.0689	4.8612
6	5	0.6417	0.0835	4.7851
6	6	0.6430	0.0539	4.9399
6	7	0.6448	0.0667	4.8726

From this we can see that the results are very similar to running a single layer, with a slight improvement in model (5, 5) with an R-squared raising to 0.0916 and MSE reducing to 4.7430 in this test. However, subsequent tests reveal this gain to be nullified by the random nature of the data, therefore we have decided that a single-layer model with five nodes would be best for further testing, as this is the fewest nodes we can use while maintaining accuracy.

We then tested this model against the data previously used to test our other models to compare results. In previous tests we trained the model one hundred times however this would take around fifty times longer for ANN, so we settled for testing each model five times instead and taking the averages from each test. The results of these tests have been added to the below table.

Dataset	Source Station	Destination Station	Accuracy	Average R-squared	Average MSE	Average RMSE
KNN3	Diss	Stowmarket	0.6108	0.0893	4.7686	2.1837
BRR	Diss	Stowmarket	-	0.0811	4.8051	2.1921
ANN	Diss	Stowmarket	0.6419	0.0676	4.6617	2.1591
KNN3	Ipswich	Manningtree	0.4120	0.0625	5.7086	2.3893
BRR	Ipswich	Manningtree	-	0.1078	5.6067	2.3678
ANN	Ipswich	Manningtree	0.3965	0.1057	5.5348	2.3526
KNN3	Stratford	Liverpool St.	0.2377	-0.0762	12.1284	3.4826
BRR	Stratford	Liverpool St.	-	0.1705	9.5594	3.0918
ANN	Stratford	Liverpool St.	0.2291	-0.0869	12.5282	3.5395
KNN3	Stratford	Shenfield	0.5221	0.1838	7.5487	2.7475
BRR	Stratford	Shenfield	-	0.0611	8.2939	2.8799
ANN	Stratford	Shenfield	0.5432	0.1564	8.1705	2.8584
KNN3	Manningtree	Ipswich	0.4547	-0.0323	6.9748	2.6410
BRR	Manningtree	Ipswich	-	0.0333	6.5167	2.5528
ANN	Manningtree	Ipswich	0.4340	0.0300	6.8938	2.6256
KNN3	Diss	Norwich	0.5029	0.0171	3.8696	1.9671
BRR	Diss	Norwich	-	0.0394	3.8515	1.9625
ANN	Diss	Norwich	0.4952	0.0342	3.7269	1.9305

As you can see, these results are all very close to the results gained from previous tests, generally within 5% of the results from the two KNN models and the BRR model. This tells us that despite the complexity of the ANN model, and the additional processing time required to train it, the results are little to no better than the comparatively simple KNN3 model.

Evaluation of the ANN Model

Through testing the ANN model, we have already found the optimum layer and node size, activation function and solver. When looking at the classification reports for some tests, we wondered if reducing our data set to a limited range of changes in delay could result in a more accurate model. This is because larger delays are far more unpredictable as they are generally the result of unforeseen circumstances such as breakdowns or customer disturbances on the line rather than more predictable small delays such as waiting at track points for another train to pass or gains made in running the train faster and stopping for less time at each station. As such we have decided to make this change into a second ANN model and if the results are notable, to compare that model to previously tested models.

ANN Model 2

We started by doing a preliminary check on a journey from Ipswich to Manningtree only looking at journeys where the delay changed forward or backward by up to 5 minutes. From this test we found that although the accuracy stayed the same, there was a noticeable improvement in the R-squared and MSE values reported, showing that the incorrect guesses given in this model were closer to the correct values than with previous models. Therefore, we decided to do another run through the dataset shown previously to see how this model compares to the previous one.

Dataset	Source Station	Destination Station	Accuracy	Average R-squared	Average MSE	Average RMSE
ANN	Diss	Stowmarket	0.6419	0.0676	4.6617	2.1591
ANN2	Diss	Stowmarket	0.6576	0.3260	0.7820	0.8843
ANN	Ipswich	Manningtree	0.3965	0.1057	5.5348	2.3526
ANN2	Ipswich	Manningtree	0.4079	0.1862	1.6151	1.2708
ANN	Stratford	Liverpool St.	0.2291	-0.0869	12.5282	3.5395
ANN2	Stratford	Liverpool St.	0.2558	-0.0730	3.4751	1.8642
ANN	Stratford	Shenfield	0.5432	0.1564	8.1705	2.8584
ANN2	Stratford	Shenfield	0.5600	0.5841	0.7609	0.8723
ANN	Manningtree	Ipswich	0.4340	0.0300	6.8938	2.6256
ANN2	Manningtree	Ipswich	0.4430	0.0748	1.1655	1.0796
ANN	Diss	Norwich	0.4952	0.0342	3.7269	1.9305
ANN2	Diss	Norwich	0.5124	0.2760	0.8227	0.9070

We can see that the change in dataset parameters to exclude changes in delays less than -5 minutes or more than 5 minutes results in a slight gain to accuracy but a clear noticeable improvement in the R-squared and MSE values across the board. From this, we wondered if the same gains could be made to the KNN model using these data sets, since the change was in our selected dataset rather than in the model itself. A brief check on the Diss to Norwich journey showed a similar gain in both accuracy and error rate as seen with the change from ANN to ANN Model 2, therefore we will check one final time with a new KNN model with this new restriction.

KNN Model 4

We applied the same datasets used in ANN Model 2 to the previously built KNN3 model to create our final KNN model. The results of testing this model alongside ANN2 and KNN3 are listed below.

Dataset	Source Station	Destination Station	Accuracy	Average R-squared	Average MSE	Average RMSE
KNN3	Diss	Stowmarket	0.6108	0.0893	4.7686	2.1837
ANN2	Diss	Stowmarket	0.6576	0.3260	0.7820	0.8843
KNN4	Diss	Stowmarket	0.6219	0.3558	0.7573	0.8702
KNN3	Ipswich	Manningtree	0.4120	0.0625	5.7086	2.3893
ANN2	Ipswich	Manningtree	0.4079	0.1862	1.6151	1.2708
KNN4	Ipswich	Manningtree	0.4196	0.1159	1.7797	1.3340

KNN3	Stratford	Liverpool St.	0.2377	-0.0762	12.1284	3.4826
ANN2	Stratford	Liverpool St.	0.2558	-0.0730	3.4751	1.8642
KNN4	Stratford	Liverpool St.	0.2650	-0.2195	3.9284	1.9820
KNN3	Stratford	Shenfield	0.5221	0.1838	7.5487	2.7475
ANN2	Stratford	Shenfield	0.5600	0.5841	0.7609	0.8723
KNN4	Stratford	Shenfield	0.5370	0.5568	0.8144	0.9024
KNN3	Manningtree	Ipswich	0.4547	-0.0323	6.9748	2.6410
ANN2	Manningtree	Ipswich	0.4430	0.0748	1.1655	1.0796
KNN4	Manningtree	Ipswich	0.4617	0.0558	1.2015	1.0961
KNN3	Diss	Norwich	0.5029	0.0171	3.8696	1.9671
ANN2	Diss	Norwich	0.5124	0.2760	0.8227	0.9070
KNN4	Diss	Norwich	0.5117	0.2172	0.8960	0.9466

From these samples we can see that this model performs comparably or better than any of the other models tested on similar datasets. Considering the KNN model requires much less processing than the ANN models for a similar or greater accuracy and lower error rate, this model seems like the natural choice for our project. We will still be creating and saving both models to our project, however.

Model Implementation

With testing concluded, we implemented the KNN4 model into our project. The model is generated ahead of time then saved into the database, stored in the network object of the rail line it is part of. When needed, it is loaded from the network object and used for prediction, meaning we do not need to build the model every time (though since each KNN4 model is relatively small at around 30,000 entries, this would not be an issue in general). For the final ANN model, we did the same, building the models ahead of time and storing them in the network object. These were more important to build ahead of time since they typically take 10-20 seconds per station to build, but the final model is only a couple dozen kilobytes.

To use the chosen model, we can call **data.process_data.user_to_query()** providing the TPL code of the source and destination stations, as well as the current delay in minutes. This function then finds which network the two stations are on, joins a path between them, finds the current date and time and calls another function for each pair of stations on the path, which itself pulls the KNN4 model from the network object to find the change in delay using **model.predict(data)**. These delays are then summed up across every set of stations, and the final delay is returned to the user.

We have found a slight issue with this model in that sometimes there is not enough data on negative train delays (where trains are arriving early to station) to accurately predict what the change in delay will be going forward, which leaves the model under-estimating by many minutes. We looked at what could be changed in the model to help with this but ultimately decided on a simple workaround of ensuring the total delay never goes below -1. Although not the best fix in our opinion, it does accurately mirror how a train would work in the real-world, in that it would usually stop at a station until the correct leaving time or be stopped at a junction to allow another train to pass if it’s ahead of schedule.

Conversation Control

Implementation

Testing

Unit Testing

Unit testing was carried out throughout the project to ensure any changes made to an individual component would not break that component, and ensure it is working correctly. For testing, we used the unittest library in Python, which comes with useful tools such as being able to check if values are within a given range, or that a piece of code if giving the correct error message when it fails. The unit tests can be found in the tests folder of the project.

The models were built and tested in Jupyter notebooks, which can be found inside the model folder. These notebooks are not used by the program itself but were included to show how the models were tested and what data we looked at collecting.

Integration Testing

System Testing

Usability Testing

Evaluation

Evaluation and Improvements to Datasets and Models

The final version of our program allows for the use of both KNN and ANN models, however KNN appears to give far more realistic estimations based on our testing. Further experimentation with the ANN model may give us more realistic estimations from real-world data, but since we proved through testing that KNN is slightly more accurate with a smaller error, we decided to settle with using KNN. We chose not to implement the Bayesian model in our final program as we could not agree to an accurate algorithm to interpret the floating-point result as a categorical value, despite many hours of discussion and experimentation. We felt this was fine, however, as the BRR model did not outperform any other model by any reasonable extent during testing. We were also concerned that the BRR model would exceed the maximum database size allowable on GitHub (100MB) due to the size of the current KNN models being held.

To possibly gain more accurate predictions, we could add the starting station for trains in the data set. Since some trains start at stations other than Norwich or London Liverpool Street (such as the weekday service from Ipswich to London Liverpool Street), they are likely not going to be delayed at the start of their journey compared to those that left at the track-end stations. This would require asking more information from the user, however, such as the origin station for the train or how many stations the train stopped at prior to the current station. We would also need to build new checks into the scripts that pull train data from the scraped CSV files. We felt that the complexity that this would add to the model and predictions would offset how useful that information could be.

Accuracy in predicting the delay in **data.process_data.user_to_query()** could possibly be improved by changing the value of the now parameter based on how much time will pass after every station, however this would require reworking the current implementation for what was felt would be little gain in real terms.

We spoke about including historical weather data into our model but after a few hours of study we concluded that it would likely not have a noticeable effect on train delays due to trains rarely being affected by a single weather event. For example, trains will continue to run when it is snowing unless the snow is deep, which would not be picked up easily from historical weather data and would account for so few delays across the measurement period that we felt it would be more likely to generate false positives than assist with correct guesses.

Summary

Due to unforeseen issues, the project took us longer to complete than we expected, however we feel our early planning allowed us to complete all the features we intended to add to the system before final submission. Although we were unable to complete much of the project by the timescales estimated on the Gantt Chart, we found that referring to that chart was still useful at our later meetings for planning around any issues and keeping the project moving forward.

At points we found that the libraries we thought would be useful during the design stage were not as useful during production. For example, when attempting to use Scrapy to scrape for train tickets, we found issues in trying to parse the scraped data and through research we found the guides and documentation for Scrapy was somewhat lacking, along with being quite confusing for new users. We decided to drop the progress made on Scrapy and switch to BeautifulSoup which, although having less functionality, did everything we needed and had far better documentation – we managed to get further in two hours with BeautifulSoup than we had in eight hours with Scrapy. We learned from this that we should not be too worried about dropping a library or a piece of work we have spent time on if we think we can get a better result using something else. This was particularly important a little later when it came to the Knowledge Engine, as we found that PyKE was also a very difficult library to use, but we managed to make great strides in development by switching to Pyknow/Experta and using some of the knowledge gained from PyKE to inform us on how to move forward quickly. We feel that using the Rapid Application Development (RAD) methodology helped us here as we could quickly switch development without tying up other parts of the system or feeling like we would need to change the requirements of the project at all, as our changes fed back into the look at user design at our following meeting.

References

Abbink, M. (n.d.) *IBM Plex Mono*. IBM Brand Experience & Design. Available at: https://fonts.google.com/specimen/IBM+Plex+Mono.

Adambro. (2007) *Scan of an example of a ticket produced by the Shere FASTticket machine at Huddersfield railway station for a journey to Honley.* [image] Available at: https://en.wikipedia.org/wiki/File:ShereFASTticket-Example.png.

Barrus, T. (2018) *pyspellchecker*. Available at: https://github.com/barrust/pyspellchecker.

GarethT, Diaz-Gonzalez, J. and medecau. (2020) *FuzzyWuzzy*. Available at: https://github.com/seatgeek/fuzzywuzzy.

Grinberg, M. (2014) *Flask-SocketIO*. Available at: https://github.com/miguelgrinberg/Flask-SocketIO.

Khan, M. (2019) *RecordRTC*. Available at: https://github.com/muaz-khan/RecordRTC.

Koumijan, A. (2015) *datefinder*. Available at: https://github.com/akoumjian/datefinder.

Livesey, A. (n,d) *Empty platforms at Piccadilly Station on Thursday.* [image] Available at: https://www.theguardian.com/uk-news/2020/mar/20/train-services-across-britain-to-be-stripped-back-from-monday#img-1.

Lord, D. (2019) *Flask.* Available at: https://github.com/pallets/flask.

Material Design. (2021) *Material Design*. [online] Available at: https://material.io/design/material-studies/reply.html#color.

Material Design Icons by Google. (2016) *Material Design Icons*. Available at: https://material.io/resources/icons/.

Microsoft. (2021) *Instant Messaging with Microsoft Teams, IM, Group Chat*. [online] Available at: https://www.microsoft.com/en-gb/microsoft-teams/instant-messaging.

Montani, I., Peters, H., Honnibal, M. and Garg, H. (2021) *spaCy*. Available at: https://github.com/explosion/spaCy.

Mozilla (2021) SpeechRecognition. Available at: https://developer.mozilla.org/en-US/docs/Web/API/SpeechRecognition.

Mozilla (2021) SpeechSynthesis. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API/Using_the_Web_Speech_API#speech_synthesis.

Piedrafita, M., Mock, W., YoonShin, P., Tea, I., Morales, R., Rauch, G. and PatrickJS. (2018) *Socket.IO*. Available at: https://github.com/socketio/socket.io.

Pradeep. (2019) *Skype Desktop app for Windows and Mac will allow you to blur your background in video calls.* MSPoweruser. [online] Available at: https://mspoweruser.com/skype-desktop-app-for-windows-and-mac-will-allow-you-to-blur-your-background-in-video-calls/.

Spacy.io. (2021) *Language Processing Pipelines.* Explosion. [online] Available at: https://spacy.io/usage/processing-pipelines.

Zhang, A. (2017) *Speech Recognition.* Available at: https://github.com/Uberi/speech_recognition.