Tweedejaarsproject BSc KI

Chatbot Gerrit

Commissioner:
UvA ICT
Bureau of Communications

Representative:
Mendel Stecher

Authors:
Tim Ottens (11147598)
Bence Tijssen (11330155)
Bram Akkerman (11322500)
Thomas van Osch (11248815)
Mattijs Blankesteijn (11318422)

Supervisor:
Victor Milewski

University of Amsterdam

June 2018

Contents

Abstract

This paper proposes a collaborative implementation on a question answering system at the University of Amsterdam (UvA). The system is built as a chatbot framework specialised in exhaustive search on a specific web domain, namely that of the UvA. The implementation of the system is split into two parts where this part focuses on the *Human Interaction* and *Information Retrieval* while the second part focuses on more domain-related extensive search [2]. A pipeline framework is proposed to make full use of the possibilities of the concerning web domain. Solutions for specific problems such as *keyword detection*, *Frequently-Asked-Question matching*, *backend connection* and *intelligent rule-based decision making* are suggested. The whole framework is deployed on a web server where the answers to the end-user's query are displayed using asynchronous communication.

1 Introduction

Human conversations are a very common appearance in daily life. Since a few decades humans no longer try to only speak with other humans or animals but with and via computers. When these computers are able to have some sort of autonomous behaviour they are called *bots*.

Human beings exhibit a high level of intelligent behaviour, whereas computers excel at staying organised and searching through their organisation. The field of *information retrieval*, that is occupied with such behaviour in computers, has been an ongoing field of research since the last century [3]. When one combines the mimicking of human behaviour with information retrieval and bots resulting in a complete conversation between both, that combination will end up in the domain of the *chatbots*, which are interactive question answering systems.

Nowadays chatbots are a common appearance on websites. Companies use these intelligent bots to help customers find their needs. Asking a natural question and receiving an answer in a human-like way is preferred over a static search bar, since people are more satisfied with an interaction than with such a one-sided search [9].

1.1 Proposed Product

The goal of this product is making a chatbot which is specialised in retrieving information by exhaustive search in a specific web domain. The problem is split in two different parts such that this report will specifically focus on the human interaction, whereas the second part focuses on the information retrieval from the web domain. The second part is performed by a different group of researchers at the University of Amsterdam (UvA) [2]. Together a question answering system is implemented with the ambition to help students every day to find their way through the UvA web domain [1]. The distribution of tasks among the two groups is visualised in figure 1.

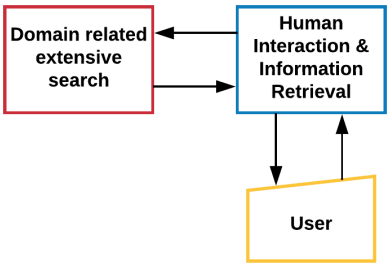


Figure 1: Visualisation of the design of the product. The red box visualises the task of the backend search using a specific way of searching the UvA web domain [2]. The implementation of the blue box is described in this paper.

Whilst the final implementation is written in a general manner when possible, this report focuses specifically on the web domain of the UvA. At the moment some parts of the current web domain are structured in a not completely logical way. This makes browsing the entire domain not an easy and obvious task, especially for new students. Instead the UvA would like to have an interactive question answering machine, which is able to understand a user's query. It should then return either a link to an internal website or a small segment of text, in which both the replies should answer the question given by the user.

This report, together with the provided product is based on earlier research [5] in which the underlying data structure of the UvA web domain has been analysed and mapped. This data structure visualises the internal structure of the UvA in a hierarchical way, namely by splitting it into different faculties and studies in a tree-based way. This hierarchical structure is used to efficiently search the websites to provide a relevant answer to the user's query.

[1] `http://www.uva.nl/en/home`

The implementation of this final product in this report is organised as follows. First of all, in Section 2.1 the high-level implementation of the chatbot will be described. Next, in Section 2.2 the main algorithm that is used internally will be explained. Thereafter, Sections 2.3, 2.4 and 2.5 discuss different methods and techniques to be able to show some of the results of the implementation in Section 3. Finally, in Section 4 everything will be discussed and suggestions for future development will be given together with the conclusion of the product.

2 Method

In this section the approach of the specified problem is described on a broad level. Section 2.1 explains the phases of the internal processing of a query. Section 2.2 describes how the implemented algorithm works, based on its classes. The next section (Section 2.3) defines the communication of the frontend and backend while Section 2.4 lists the different techniques used and why these specific techniques are chosen. Finally, Section 2.5 links the Python framework to a website interface.

2.1 Chatbot Logic

For the product an extensive flowchart has been established that can be seen in figure 2. The flowchart shows the different stages in the chatbot's program. Following subsections contain high-level descriptions of each process from this flowchart.

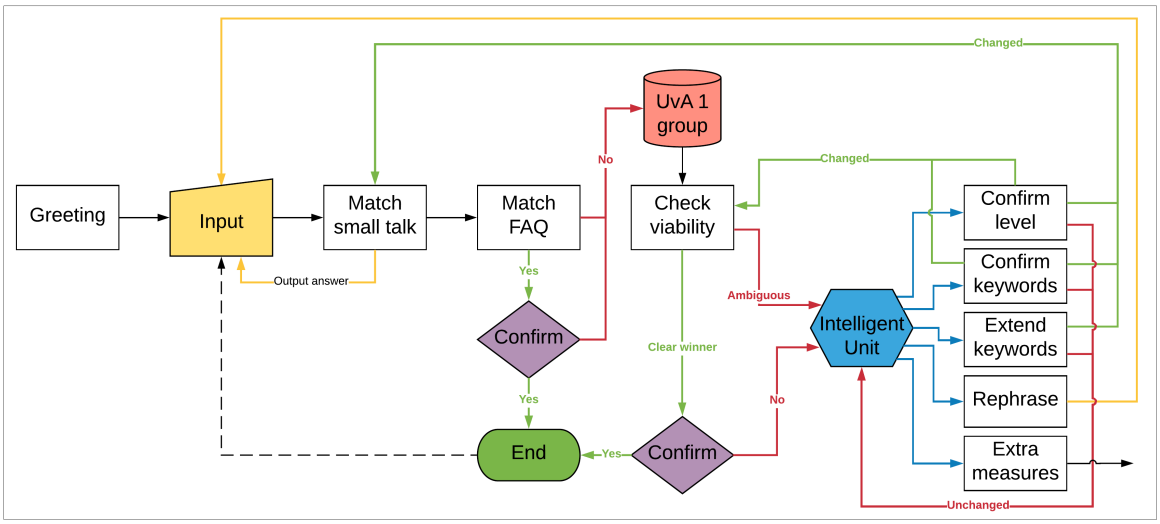


Figure 2: The internal chatbot logic. Arrows represent moving from one phase in the program to another, and when multiple arrows point out of one box they represent different outcomes of a certain phase of the program.

2.1.1 Input and Small Talk Phases

The chatbot starts off by *greeting* the user. At this time, the user can respond by sending input to the chatbot, represented by a yellow box in the flowchart (see figure 2). Language and keywords will then be extracted from the input. The user may then chat with the chatbot about subjects unrelated to their search of the UvA website in the *small talk* phase. The answer to the user query should then be matched with preprogrammed answers in the chat database (see Section 2.4.3 for details on the method used for *small talk matching*). Note that in the current implementation the chat database is very limited. At some point, the user will then ask their search-related question, which will result in the continuation to the *FAQ matching* phase.

2.1.2 The FAQ matching phase

In the *FAQ matching* phase the chatbot will try and match the user's input with the Frequently Asked Questions (FAQ) collected from the UvA website. If a sufficient match has been found, the user is asked to confirm that the question found in the FAQ matches their question. If so, the answer to the question is presented to the user. If the user finds the answer satisfactory, then the goal state is reached. If the user doesn't find the answer to the question helpful, if the user wasn't looking for the question proposed or if no matching FAQ was found in the first place, the *backend* phase follows.

2.1.3 Backend Phase

The *backend* phase, which is represented by a red box in figure 1 and a red cylinder in figure 2, is where the user input is send to the backend. A list of possible matching answers is returned whereafter a viability check is executed in order to test whether there is a clear winner [2]. In that case the winner is returned to the user and confirmation is asked. If there is no clear winner, or if there is a negative confirmation the list with options continues to the intelligent unit.

2.1.4 Intelligent Unit

When the is no clear winner in the list of documents returned by the backend, the intelligent unit (IU) will make a decision as to which follow-up action to choose. This IU is a rule-based system that can choose one of five options.

Level confirmation. When this option is chosen the user is asked to confirm the level of their question. The level is the study or faculty that the input question is related to. If a level was already guessed in previous phases the user is asked to confirm it, while if no such guess exists the user is asked to input it themselves.

Keyword confirmation. This step is chosen when among the top options returned by the backend different answers include different keywords. The user is asked to confirm one of the keywords. If the keyword is confirmed, the answers not containing that keyword are eliminated. However, if the keyword is unrelated to the user's search, the answers containing the keyword can be eliminated. When a keyword has been confirmed, it is marked as such in order to avoid asking about it again. In this phase, it is possible that the right answer is accidentally eliminated due to an answer not actually containing a related keyword. More on this in Section 3.2.2. Note that this issue is mitigated by the next phases.

Keyword extension. This option is chosen when the previous two options are no longer desirable. The user is asked to come up with a single keyword that they believe to be important to their search. The keyword is then marked as confirmed, and the original input, now including the keyword, is sent back to the point after where the original input is given. This happens to be the matching of the chat database.

Rephrasing. This option is not desirable, and will only be chosen at most once per conversation, and late into the conversation. When invoked, this option will ask the user to write their original question again, this time using different words. Previous information about the conversation will be kept.

Other measures. This option is a last resort, when nothing else seems to work. It ignores a lot of the previously gathered information, and makes wild guesses. If some other step in the process of the IU resulted in the right answer being discarded, this measure will ensure that it will eventually be asked. If the answer was not among the options of the backend at all, the rephrase step will be repeated, but this time without keeping any previous information. Any incorrect piece of information that might have been causing wrong answers is hereby discarded.

2.2 Algorithm

The algorithm used for the implementation of the chatbot is strongly based on the logical structure described in the previous Section. In this Section the implementation of the algorithm is described. Since an object-oriented approach was followed, it makes sense to discuss parts of the program per class.

The following subsections will follow this structure. How these classes are linked together on the server is discussed in Section 2.5.2.

2.2.1 The Main Class

The Main class creates an object on which the `run()` method can be called once. When it is called, `run()` will start the main loop of the algorithm, which will run until the user ends the program. The Main object will store a Conversation, Chatbot and IU object.

Once the program is running, it starts at the input phase. Each phase with incoming arrows in the process flowchart as seen in figure 2 has its own function so that the algorithm can continue from that phase onward. As the program runs through the algorithm, different phases are called based on the structure in the aforementioned figure. The IU object stored by this class and the answers of the user dictate which phase is chosen when multiple options are available.

How exactly the `Main` class is implemented in Python can be found in Appendix A.

2.2.2 The Conversation Class

The Conversation class stores and processes the user's input. This includes extracting keywords and level from the user's query, and turning available information into a usable format for the backend. A Conversation object exists until the user's query has been answered. When a new question (not small talk) is asked, it is renewed. The Conversation class uses the Sentence class to store individual sentences with their properties and to handle the actual processing.

2.2.3 The Chatbot Class

The Chatbot class handles user interaction and the calling of processes that can lead to answers (chat and FAQ matching, backend). It gets created exactly once at the initialisation of a Main object, and can therefore store a specific user's language and session ID.

2.2.4 The IU Class

The IU class stores a list of dictionaries returned by the backend, where each dictionary represents a possible URL. The IU's tasks all have to do with this list. It can choose a line of questioning based on this list, update the list either directly or by specifying a level or keyword and it can also check if there is a URL worth returning to the user.

Its duration is the same as a Conversation object, that is to say, as long as it takes to correctly answer one of the user's questions about the UvA website. In order to perform its tasks during this time, it keeps track of whether the level has been confirmed by the user, which keywords have been confirmed by them, which URLs have been marked by them as incorrect and its own history of choices.

2.3 Backend Interaction

This Section describes how the communication with the backend phase is performed in the current implementation. The interaction is divided in input and output to keep the connection as clean as possible. The input is discussed in Section 2.3.1, the output in Section 2.3.2 and their combination in Section 2.3.3.

2.3.1 Input backend

The chatbot communicates by means of a *dictionary* with the database delivered by the backend [2]. A dictionary was chosen due to its functional mapping property where different pieces of information about an incoming query can be structured. The properties of this dictionary are visualised in figure 3. Due to the bilingual implementation (Section 2.4.1), it is important to send information on the queries `Language` in order to search the web domain in the correct locations. To identify the type of answer a user is looking for the `Type` of the query should also be given to the backend. Because there are different levels in the UvA web domain, to enable optimal search the search engine should be aware of the `Level` of the user's query [5]. To enable intelligent question asking on top of the information on the original user query, it is important to send `Keywords` on the query which can be modified during the answering process. For backend implementations with natural language processing included it can be useful to know the original users query, named as `Source` in figure 3 [2].

Language	INT: $0 = Dutch$, $1 = English$
Type	INT: $0 = unknown$, $1 = who$, $2 = what$, $3 = where$, $4 = why$, $5 = when$, $6 = how$, $7 = which$
Level	STR $< study > \vee < faculty > \vee$ "UvA" $\vee$ $None$
Keywords	SET $\{< keyword_1 >, ..., < keyword_n >\}$
Source	STR < original sentence >

Figure 3: Outgoing data structure. This is defined as a dictionary with specific information about the end-users query.

2.3.2 Output backend

The database responds with a list of n dictionaries [2]. The properties of these dictionaries are described in figure 4.

URL	STR: < url >
Score	FLOAT: < number between 0 and 1, 1 equals a complete match >
Level	STR $< study > \vee < faculty > \vee$ "UvA" $\vee$ $None$
Keywords	SET $\{< keyword_1 >, ..., < keyword_n >\}$
Answer	STR < passage from the HTML article >

Figure 4: Incoming data structure. The return value consists of a list with variable length of these structures.

The end-user is given an answer in natural language, alongside with the original link where the information was found [2]. To let the chatbot give this information, the information should also be received from the backend. Therefore, the dictionary of a possible answer should contain the original `URL` and an `Answer` in natural language. Because the output is defined as a list of dictionaries, each dictionary comes with a `Score`, which also enables more advance intelligent handling as described in Section 2.1.4. In cases where the answer might be found on a different level as intended by the end-user, the correct answer should also be possible to answer even though it is not exactly the correct level. Therefore the `Level` on which the answer was found is also important to receive. Lastly, to guide the end-user faster and more reliable to a final answer the `Keywords` of a document are received in order to enable questioning about these words.

This whole list is sorted by the values of every *Score* key of each of the dictionaries. That gives a list:

$$[D_1, ..., D_n] \text{ where } \forall_k \ (0 < k \leq n) : D_k['Score'] \geq D_{k+m}['Score'] \text{ for } 0 \leq m \leq n - k$$

2.3.3 Combination

By defining the input and output as above strong version-independence can be guaranteed. As long as the data structures send to one-another stay the same the actual handling on both sides can be updates without affecting the application as a whole negatively.

This type of definition also guarantees that the given output always returns the best matching option as the first dictionary while it still enables the chatbot to discard that option in case of a low score of confidence.

2.4 Techniques

To implement the described framework different techniques are needed for different purposes. This Section describes techniques for *handling multiple languages* (2.4.1), *keyword detection* (2.4.2), *chat functionality* (2.4.3), *FAQ matching* (2.4.4), *domain scraping* (2.4.5) and *study recognition* (2.4.6)

2.4.1 Handling multiple languages

Students and teachers from around the globe visit the website of the UvA. The website therefore is built for two languages: *Dutch* and *English*. The chatbot needs to handle both of these languages. Using the Natural Language Toolkit (NLTK) an procedure to recognise the language of an input string is designed [8]. By checking a corpus of English words together with a certain threshold, the language is chosen. Below the threshold, which is defined in the configuration file, meaning the sentence has not a high enough confidence level to be classified as English, the language gets classified as Dutch. The chatbot does this for all user input that isn't an answer to a specific question. In addition the language can be changed by interacting via the user interface (Section 2.5).

2.4.2 Keyword Detection

```
function keywords(sentence):
    tokens = set('NOUN', 'ADJ', 'NUM')
    language = CURRENT_LANGUAGE
    tokenized = TOKENIZE(sentence)
    tagged = TAG([x for x in tokenized if x not in STOPWORDS(language)])
    RETURN [word for (word, token) in tagged if token in tokens]
```

Algorithm 1: Keyword Detection. Using the current language a sentence is tokenized by NLTK [8] and filtered on the open class PoS tags *nouns*, *adjectives* and *numbers*.

In order to understand the user's query the chatbot searches the query for relevant information. 'Relevant' parts of the users query are defined as *keywords*. The goal is therefore to remove stop words and other trivial words which result in a query with only descriptive and useful keywords. With these words remaining it is possible to match and search the information of the UvA websites using information retrieval. The algorithm used to detect these keywords is given in algorithm 1.

Using the *NLTK* function `word_tokenize` the query string is split up, uppercase characters are converted to lowercase characters [8]. With the words separated `nltk.pos_tag` assigns each word a Part-of-Speech (PoS) tag [8][4]. Not all words are relevant to the semantic of the query. This product therefore only remembers the universal PoS tags *noun*, *adjective* and *numbers*. Only the universal tagset of *NLTK* is used [8], since it includes all these PoS tags. Words with a different PoS tags are discarded, including punctuation. The remaining words are then sent to be matched with the UvA information.

This process of parsing the input sentence with PoS tags is applied to both English and Dutch queries. The tokenization is indifferent; punctuation in both languages is almost the same. After recognising the language, or being selected by the user, *nltk.pos_ tag* takes the language as argument. The support of the Dutch language of *NLTK* is not optimal and therefore the accuracy of the words with the correct PoS tag is not as high as the accuracy of English. However, for the scope of the production both languages are tagged by *NLTK*. See Discussion Section 4.2 for further improvement. As a temporarily solution a list of Dutch and English stop words are added to the process. The list contains common words which are filtered out of the query. After this process, only the relevant keywords of the query are left and matched.

2.4.3 Baseline of Chat Functionality

In order to give the end-user a better dialogue-experience, some basal dialogue questions are stored with their answers in a YAML Ain't Markup Language (YAML) database as described in Section 2.1.1 [10]. An example of how these questions and answers are stored is given in table 1. These questions are matched with the input questions as follows:

$$\{keyword_1, ..., keyword_n\} \subseteq (\{word_1, ..., word_m\} \cup \{\texttt{CONFIG\_LIST}\}) \text{ with } n \geq 1 \text{ and } m \geq 1$$

Where `CONFIG_LIST` is a list defined in a config-file. The contents of that list are considered to be ignored in the procedure of matching sentences. Currently, some example words from the list are *Gerrit* and *You*.

Notice the contamination in the second sentence of Table 1. It is there to provide a more robust matching with the input question. If list m provides more relevant words, the chance of matching the input keywords with the actual correct question is much higher.

Question	Answer
Which studies do you know?	`I know every study one can study at the University of Amsterdam.`
How far is the distance to the sun?	`The sun is about 93 million miles from earth.`
Yes	`Great`

Table 1: Examples of YAML questions

Moreover, to improve the user-experience even further the chatbot uses some more YAML files to keep track of different ways of human sentence-expression. For example a list is saved containing different ways of confirming a question because *'Yes'* is not the only manner of confirmation in the English language. The same database is also used to enable the chatbot answering in good natural language sentences. These sentences can contain variables to fill in later, for example:

"And here's a {link} to the page where I found information on {subject}."

Where *link* and *subject* are variables that can be filled according to the conversation.

2.4.4 FAQ Matching

In *Match FAQ* the query of the user is compared to questions on the UvA website that are in the FAQ list. This step prevents the chatbot, in some cases, from doing an exhaustive search on the UvA website. First a Term Frequency-Inverse Document Frequency (TF-IDF) table is made with the keywords of all questions in the FAQ and the user's query [17]. A TF-IDF table represents the importance of keywords based on their occurrence in a document. After creating the table, cosine similarity is used to determine the score of the matched question of the FAQ with the user's query. If the query of the user is matched with a question on the FAQ list, then that answer is returned to the user with an additional confirmation question to determine that this information is what the user was looking for. When the user responds with "yes" the chatbot returns to the input stage in the flowchart and asks if the user wants to ask anything else. In the case of a "no" to the confirmation question, information about the user's query is send to the backend *(UvA ICT 1)* where the information will be used to acquire answers for the user from the UvA website.

2.4.5 Web Domain Scraping

For computational reasons different important aspects are scraped from the UvA web domain. In all cases there exist two final files due to the bilingual implementation.

First of all, to match with local databases the FAQs with their *HTML* coded answers are scraped from the UvA web domain and saved locally. Furthermore a list of studies and faculties is scraped and matched with the internal list of the backend. Lastly, to enhance user experience all list of abbreviations of studies and faculties is taken from the study-related *URLs*.

2.4.6 Study Recognition

Using the keywords of the user's query, the baseline of the question can be extracted, to better specify the input query. To do so, the algorithm checks for every study or faculty what the similarity is with the input of the user. For this matching algorithm the biggest obstacle is typographical errors within the input query [7].

Naive matching. The naive matching approach the algorithm checks for every keyword if it appears letter by letter in the name of the study or faculty. This approach is suboptimal because the user cannot make typographical errors in their query. On the other hand, this is a computationally easy task, resulting in little delay for matching the query with studies.

Double Bigram matching. When using this approach the algorithm checks for all possible incremental combinations of the words in the list of keywords the similarity with the study. For every character in the combination it checks whether the character of corresponding or surrounding indices in the name of the study, with an index error maximum of 1. This way the typographical errors of the user get filtered out. The downside of this approach is that the list of keywords only contains specific types of tokens, but the study names consist of all types of tokens. Therefore, every token type that is not used as keyword will disrupt the score of similarity. Because of this, the error is too big to be a consistently good method of study matching.

This error can be prevented by getting the keywords of the studies before comparing, but the PoS tags on a short sentence has errors too, so not all valuable words of the study name will be preserved. Another way of tackling the error is using the whole input for checking the similarity, which will increase the amount of combinations, therefore increasing the delay for matching the query with studies.

2.5 Online Implementation

To enable a scalable implementation of the whole framework a first attempt to interact online with the chatbot is made, which will be described in this section.

2.5.1 User interface

To enable nice user interaction, a simple user interface using HTML and CSS is provided. The interface is already responsive as visible in figure 5.

(a) Desktop **(b)** Mobile

Figure 5: The Chatbot User interface on different platforms

To enable easy language switching for the end-user two country flags are visualised to indicate the current language and to enable switching. The user might want to switch manually in the unlikely case that the language detection fails, or when they don't understand the first message from the chatbot. The UvA-logo changes on language change to the correct language and scales width the width of the view port.

2.5.2 Hosting

With the comprehensive chatbot framework written in Python, the chatbot needs to interact with users over the internet via a web interface. A special network using asynchronous multithreading is used to handle different interactions with the chatbot at the same time as efficiently as possible. To do so, the Python framework is hosted on a website using *Flask* [14], a lightweight web framework using *Jinja2* and *Werkzeug*. The object-oriented Python files are called from *Flask* and communication with the client side is done with *Flask-socketIO* [6]. An index is created with *HTML*, *CSS*, *jQuery* and *JavaScript*.

As shown in figure 6, the clients directly communicate with the asynchronous socket on the server, which is triggered by different *jQuery*-events. Upon opening the website, a new thread is started for the visiting user or client. To distinguish each client, a unique session id is used which is saved together with the thread in the socket. So for each session-id (SID) a *Main* chatbot class instance is linked to it, containing the chatbot framework. The *Main* instance communicates with different chatbot classes as described in Section 2.2. This results in the chatbot only interacting on the designated client's web page. By using an asynchronous setup with *socket.IO* the index only has to be loaded once, while the socket interacts with *jQuery* on the client to show new messages. This results in a more efficient way of interacting with the user.

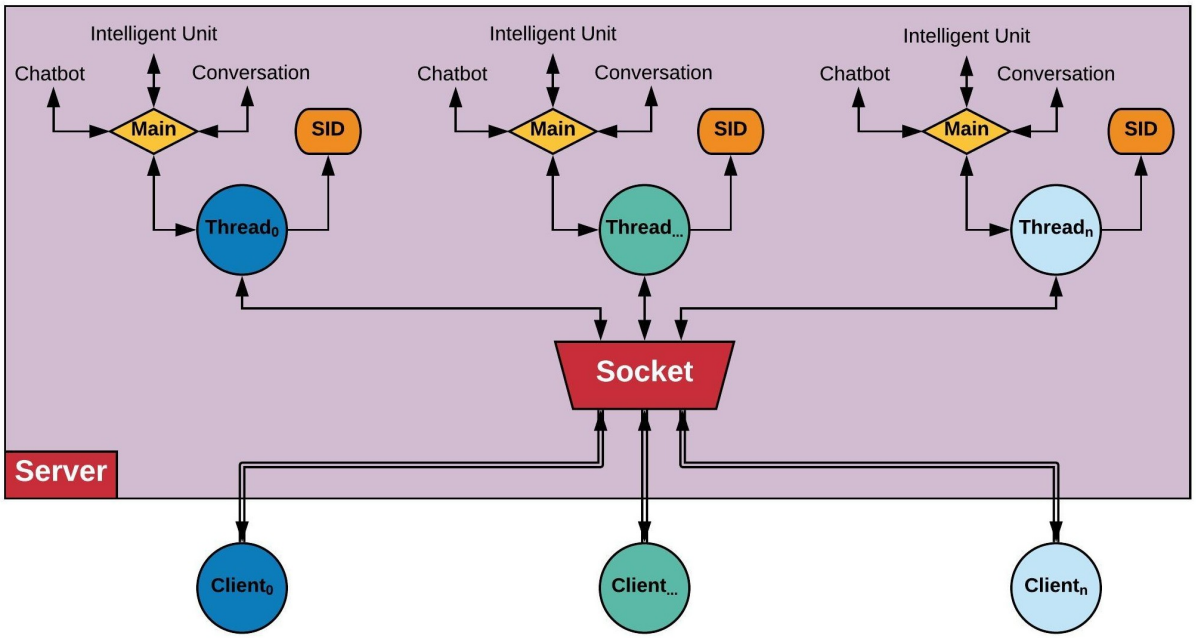


Figure 6: The chatbot hosting

3 Results and Discussion

In the following Section the results and limitations of the software are discussed. The results, embodied by the behaviour of the chatbot, are discussed in section 3.1. A non-exhaustive list of the chatbot's limitations at the time of writing can be found under Section 3.2.

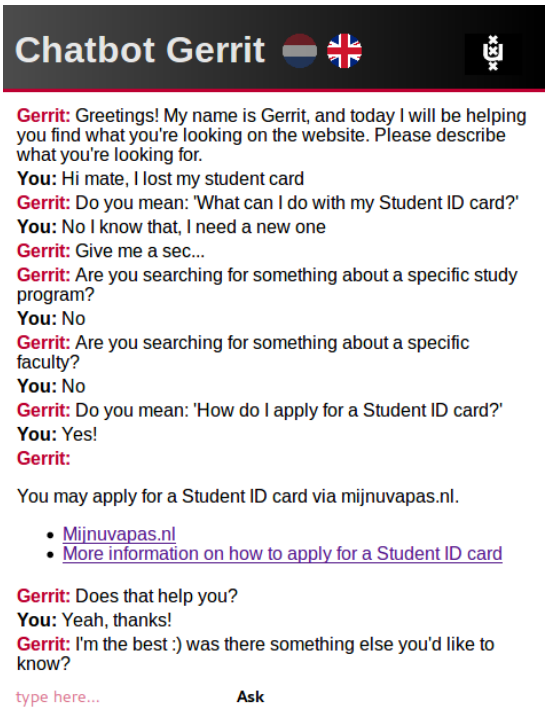
3.1 The Chatbot's Behaviour

The chatbot is specifically designed for searching the UvA web domain. It is possible to have some small talk with the chatbot, although that behaviour is limited. Currently the chatbot is able to match all FAQs within a given distance to other URLs from the general FAQ in both English and Dutch. The current backend implementation is trained on the general UvA A-Z information [2]. Improvements on more information there would also result in improvements in the knowledge of the chatbot.

A good example of the current state of the chatbot is given in figure 7a. A student lost their student ID card and does not know what to do. The chatbot matches their question with an FAQ question and asks for confirmation. When the user answers negatively their search is processed by the backend. The backend has no clear winner, so the IU decides to ask information on the level of the question, which suggests there may be multiple pages on student cards on different levels in the UvA domain. After confirming the level it does a second match with the FAQ and in this case there was a second best FAQ answer above a certain threshold. That gives the student enough information on requesting a new student ID card.

Another example of the chatbot's behaviour is given in figure 7b. A student is searching for a WiFi connection which is incorrectly classified as a Dutch sentence. This might be due to the fact that *Gerrit* is a Dutch word and *WiFi* is not already in the NLTK corpus [8]. Backend results do not have enough confidence for a clear winner to be returned already, therefore the Intelligent Unit is again searching for the level. The student is also able to have a conversation in Dutch and gives the chatbot a rephrase of their question. After some confirmations of some keywords the final answer given by the chatbot is indeed the correct one.

These examples illustrate that the chatbot is able to give correct answers within a few questions. However, there is still room for improvement here. The chatbot should for example never be able to ask if the user is searching for something about *'hi'*. To counteract such mistakes, the behaviour of the chatbot can be modified by changing the parameters of the configuration file. This can make the chatbot



(a) The final information for this student is indeed found on the last suggested page.

(b) Wifi example with incorrect language switching.

Figure 7: Example conversations with Gerrit. Results are generated with default configuration.

more or less strict about decisions. More examples that showcase specific techniques can be found in Appendix B.

3.2 The Chatbot's Limitations

3.2.1 Spell Check Matching

Currently, by using keywords a considerable amount of text can already be misspelled while some misspelled keywords still give good results.

However, to tackle the obstacle of typographical errors more deeply, one could use a spell checker. The package *Autocorrect* [11] provides a module *spell*, which checks the spelling of the input. However, using *spell* requires a considerable amount of computational power, which is not preferable in a real-time question-answering interface.

3.2.2 The Intelligent Unit

The IU assumes complete user honesty and accuracy in all but the extra measures phases. If the user mistakenly confirms a keyword or level, only the extra measures step has a chance to get the answers back that they removed. There is no option for the user to take something back, short of restarting the program.

Another issue with the IU is that it assumes the information on the UvA web pages is complete and accurate. This is not always the case. The right information might be in unexpected places, leading to the IU removing such information for being on a page under a certain study or faculty other than the one the user is looking for. An example of this is the a-z pages of certain bachelors having general information on matters like parking, the laptop help desk and Canvas. For a certain topic, any one of these pages in different a-z lists might be correct to return. However, these pages will not be returned due to seemingly being specific to their respective studies. The right information might also not include the same keywords as the search query. This could lead to the IU removing this information.

3.2.3 The Chat Capabilities

The conversation between the user and the Chatbot starts, most of the time, with a greeting upon which the user can ask some general questions to the Chatbot. When there is no related general question/-greeting to the user's query the Chatbot proceeds to the backend with the hopes of finding an answer for the user. The problem starts to arise when the user gives a general greeting or asks a general question but the Chatbot can't find a correlated greeting/question in the general questions/greetings file.

Question	**Answer**
I am <name> and I am studying AI	`Let's check`
	`Is AI an important part of your question?`
No, I haven't asked it yet	`Is your question about studying?`

Table 2: Example of a mistake in the chat capabilities

It then goes to the backend where it is possible that a positive answer is returned by mistake. When this happens the Chatbot will ask the user follow-up questions on the user's original question even though this was a general question. An example can be seen in table 2. This does not happen often but for now this could simply be resolved by adding a restart button or command.

3.2.4 Information Retrieval Sources

The only FAQ that is currently being used is the student FAQ[2] for both Dutch and English. This can be expanded to the FAQs of all levels (faculties, studies etc.). The increased levels of studies and faculties would result in direct answers on FAQ-related questions from all levels and faculties. This would improve the efficiency of the Chatbot because users would get a more detailed answer and there would be no need for an extensive search in the UvA database.

4 Conclusion

This section gives some concluding remarks on the project described in this paper. Moreover, some future implementation suggestions and recommendations specific for the commissioner are given.

4.1 Concluding Remarks

In conclusion, a first approach in making a dialogue system for searching the UvA web domain has been completed and is in use. The dialogue system is implemented as a chatbot which has been built according according to a newly designed pipeline framework.

During the implementation of this question answering system different techniques were used. First of all, a bilingual implementation has been proposed to ensure communication capabilities for most, if not all, of the target audience of the UvA. Moreover, by using part-of-speech tagging keyword detection is employed to get the most important parts of the user's query. Basic chat functionality for small talk is provided alongside FAQ question matching based on *TF-IDF* measures [17]. Lastly, using *Flask* and *socket.IO* a first scalable approach is made for online implementation of the framework as a whole.

All this frontend human interaction and information retrieval is based upon a search structure built specifically for the UvA [2]. By sending specific data structures on the information of the conversation and the user query, version-independence on both sides can be guaranteed, as long as the structures and types of its contents stay the same. Using an intelligent rule-based system different choices on handling the answers on the users query can be made, where the interaction with the intelligent system, the conversation and the chatbot in- and output is all handled by a main algorithm.

[2] `http://student.uva.nl/en/faq/faq.html`

4.2 Further implementation suggestions

4.2.1 Natural Language Processing

Understanding a user's query is and always has been a difficult task, even for artificial intelligence. With many factors influencing the semantics of a query, it is essential to take into account all of these elements. This product, as mentioned in Section 2.4.1 and Section 2.4.2, uses *NLTK*, but support for different languages other than English is limited because of this [8]. Alternatives are for example *spaCy* which creates a language model and is relatively easy to update these models [1]. Also it is possible to train the language model to fit the data better [1].

4.2.2 Self Improve Matching

FAQ matching can be improved using neural networks with feedback received from the user. Currently log files are being created from every conversation of one of the developers with the chatbot. If the information given by the chatbot was relevant to the user, the query can be matched to the FAQ question. Neural networks can then detect the significant parts of the matched sentences and create nodes with weights based on this information. With every different sentence match the FAQ matching becomes better. This is not yet implemented because not only the algorithm is of importance but also the mechanism of the feedback resulting in the storage of the queries and their FAQ match. The neural network also has to train after a certain amount of queries to update the weights which it preferably does by itself. These problems combined are the reason for the absence of neural networks in the current matching implementation.

4.2.3 Accuracy measure

The quality of the chatbot is subjective. Nevertheless, there are ways to gauge its performance. To have clear indications on the chatbots performance one of the following ways of measuring could be implemented.

First of all, one way would be to ask the user for feedback on the quality of their conversation with the chatbot. This kind of feedback can be very useful but has limitations in the amount of human work needed.

Another way would be to count the number of confirmed responses versus the number of rejected responses in general in a given conversation or over a given period of time

A third way of concrete measuring, could be counting the number of follow-up questions needed to answer a question, with each follow up question having an increasingly high weight. If a question isn't found at all, this has the maximum value, which is equal to an arbitrary amount of follow-up questions. More than that amount of follow-up questions also results in the maximum value.

To compare one of the above mentioned performances, a selection of other chatbots could be made as a baseline. A comparison is then made to the average of these results. This should give a good idea of the chatbots performance relative to other chatbots. However, one should keep in mind that the most chatbots are made for a specific goal, so that they are most of the time not one to one comparable.

4.2.4 Parameter Optimisation

The parameters in the `config.py` file change the behaviour of the chatbot. It includes thresholds that can lead the chatbot to be more or less risk-seeking. These parameters have been set to values that seem to work reasonably well, but have not been optimised. There are many optimisation techniques that could be employed to increase the chatbot's accuracy [13].

4.3 Recommendations

The product has seen desirable results regarding returning a relevant answer to a given input by a user. By implementing and using existing libraries and modules together with organised object-oriented programming the software is easily built further upon, either by pushing the known limitations or by implementing new features. For the implementation the following recommendations should be taken into account.

First, the chatbot in its current implementation is ready to be implemented for daily use, given that the implementation is done on a server with enough computational capabilities.

Secondly, as mentioned in Section 2.5.2, the framework used to run the Python code from is *Flask*. Initially *Flask* is suitable for lightweight program. However, by extending the current project to improve efficiency and obtain more satisfactory results, it is recommended to embed Flask in a WSGI server [3] or use a framework which is more capable of managing bigger projects, for example *Django* [16]. This project utilises a Raspberry Pi 2 for hosting. With the amount of daily visitors on the UvA websites it is strongly recommended to dedicate an independent server for the chatbot, one which is more capable of handling multiple visitors at once.

Finally, there may be some unknown issues with the implementation in its current state. The current implementation writes anonymous logs of its conversations in order to more easily address these issues. Some known issues are already described in Appendix C.2 and should be addressed in time.

When taking the mentioned points above into account, the implementation proposed in this paper could already be used in real-world applications.

[3] `http://flask.pocoo.org/docs/1.0/deploying/`

References

[1] Explosion AI. *spaCy Industrial-strength NLP*. URL: https://spacy.io/.

[2] Dorian Bekaert et al. "UvA ICT: Building a Chatbot". In: *UvA Second Year Project BSc KI* (2018).

[3] Nicholas J Belkin et al. "Interaction with texts: Information retrieval as information seeking behavior". In: *Information retrieval* 93 (1993), pp. 55–66. URL: https://pdfs.semanticscholar.org/781d/1cb85b0cb9d4ecdd8c1aee171a57cd5b0008.pdf.

[4] Universal Dependencies Contributors. *Universal POS tags*. Version 2. URL: http://universaldependencies.org/u/pos/all.html.

[5] Kim Gouweleeuw et al. "UvA Search Engine". In: *UvA Second Year Project BSc KI* (2017).

[6] Miguel Grinberg. *Flask-SocketIO*. URL: https://flask-socketio.readthedocs.io/en/latest/.

[7] Petteri Jokinen, Jorma Tarhio, and Esko Ukkonen. "A comparison of approximate string matching algorithms". In: *Software: Practice and Experience* 26.12 (1996), pp. 1439–1458.

[8] Edward Loper and Steven Bird. "NLTK: The Natural Language Toolkit". In: *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*. ETMTNLP '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 63–70. DOI: 10.3115/1118108.1118117.

[9] Clifford Nass et al. "Can computer personalities be human personalities?" In: *International Journal of Human-Computer Studies* 43.2 (1995), pp. 223–239.

[10] Ingy döt Net, Clark Evans, and Oren Ben-Kiki. *YAML (YAML Ain't Markup Language)*. URL: http://yaml.org/.

[11] Peter Norvig. *Autocorrect*. URL: http://norvig.com/spell-correct.html.

[12] Fernando Pérez and Brian E. Granger. "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: http://ipython.org.

[13] Duc Pham and Dervis Karaboga. *Intelligent optimisation techniques: genetic algorithms, tabu search, simulated annealing and neural networks*. Springer Science & Business Media, 2012.

[14] Armin Ronacher. *Flask*. Version 1.0.2. Apr. 1, 2010. URL: http://flask.pocoo.org/.

[15] Guido van Rossum. *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, 1995.

[16] Adrian Holovaty en Simon Willison. *Django framework*. URL: https://docs.djangoproject.com/en/2.0/.

[17] Suzanne Wetstein. *Designing a Dutch financial chatbot (Master Thesis)*. July 14, 2017. URL: https://beta.vu.nl/nl/Images/stageverslag-wetstein_tcm235-851825.pdf.

Appendices

A Main Algorithm Explanation

Here follows some basic information of how exactly the main algorithm is implemented in Python code. For explanations of individual functions, see the comments in the code.

The main algorithm uses three objects to get, store and process its data: a `Chatbot` object (referred to as `cb`), an `IU` object (referred to as `iu`) and a `Conversation` object (referred to as `conv`).

The Main class has no functions that should under normal circumstances be called from outside of the class other than `run()` (and possibly its "get" functions). This `run()` function can only be called once per instance of `Main`. It greets the user and start off the algorithm.

The following functions of the main loop are parts of the main process. Each function represents part of the algorithm from a phase from figure 2 with more than one incoming arrow up to (but not including) the next phase with multiple incoming arrows.

- `continue_at_input()` continues at "Input".
- `continue_at_chatter()` continues at "Match small talk".
- `continue_at_viability_check()` continues at "Check viability".
- `continue_at_iu_choice()` continues at "Intelligent Unit".

All these functions have multiple functions underneath them to perform parts of their objectives. Once one of these functions has performed its tasks, it will call one of the others. If at any point during this an answer is returned, `end_conversation()` gets called. If the user then confirms that the right answer was found, they are asked whether they want to ask another question. If they do, new `conv` and `iu` are set and the program is restarted from `continue_at_viability_check()`. If they don't, `end_conversation()` returns `True`. This will result in every previous "continue" function returning `True` as well, which ends with `run()` doing the same. This ends the program, and the `Main` instance that `run()` was called on should not be used from that point onward.

B Example Conversations

In this Section, some example conversations are shown to illustrate the final working of the chatbot. The results are replicable on the chatbots `current website`[4] with exception of the random generated sentences.

Figure 8: Small Talk (Section 2.1.1)

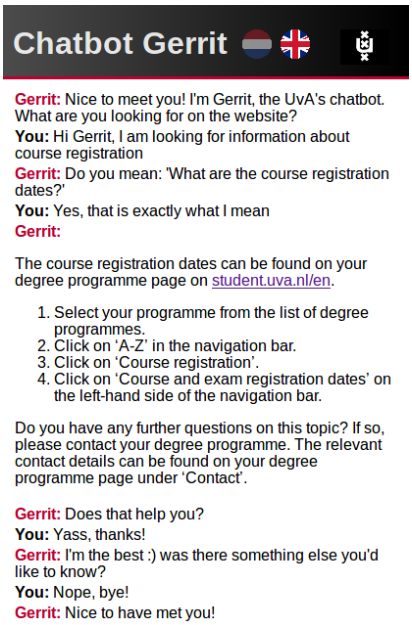


Figure 9: FAQ (Section 2.1.2)

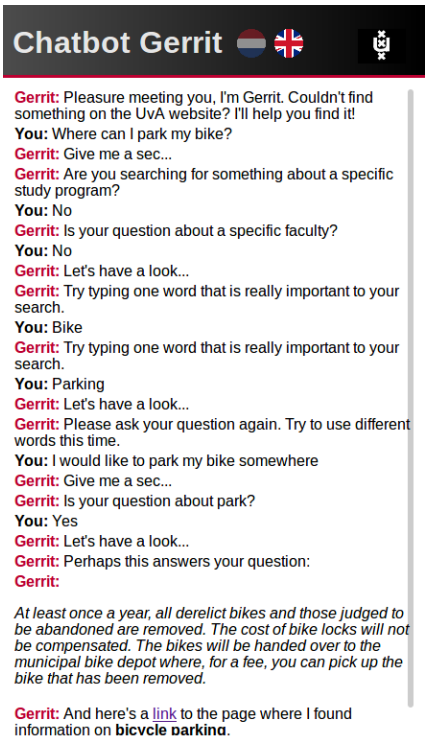


Figure 10: Intelligent asking (Section 2.1.4)

Figure 11: Backend interaction (Section 2.1.3)

[4]Self hosted first implementation: `http://gerrit.onthewifi.com`. Due to the self hosting their might be connection problems or slow responds.

C Implementation

C.1 Files

The functionality of the chatbot consists of the following files.

`run.py`	The main file to run, responsible for setting up a *server*, a *socket* and for keeping track of different *sessions*
`main_algorithm.py`	The file with the `Main` class (Section 2.2.1), responsible for using the chatbot logic as described in Section 2.1
`conversation.py`	The file with the `Conversation` and `Sentence` class (Section 2.2.2), responsible for the whole conversation
`chatbot.py`	The file with the `Chatbot` class (Section 2.2.3), responsible for user interaction
`intelligent_unit.py`	The file with the `IU` class (Section 2.2.4) responsible for choosing the next action for the chatbot
`search_faq.py`	The script responsible for searching Frequently-Asked-Questions matching (Section 2.4.4)
`config.py`	The file with some important parameters of the chatbot
`additional_en.yml`	The YAML database with some English questions and answers as described in Section 2.4.3
`additional_nl.yml`	The YAML database with some Dutch questions and answers as described in Section 2.4.3
`core_en.yml`	The YAML database with some generic English ways of sentence behaviour as described in Section 2.4.3
`core_nl.yml`	The YAML database with some generic Dutch ways of sentence behaviour as described in Section 2.4.3
`faq_neural.py`	A start for future matching expansion of the bot using Neural Networks (Section 4.2.2)

For implementation one should use the `README.md` file in the repository that should be attached to this file. On top of these files, there are some extra python files or notebooks containing scraped web domain data (Section 2.4.5) [12] [15]. Moreover, there are some files containing HTML, CSS and some images for the user interface (Section 2.5.1).

C.2 Specific Known Issues

C.2.1 Client-Server Unicode Handling

Something inside socket.IO is not correctly parsing Unicode:

- Client log before sending: ë
- Server log input: Ã«

This was an issue in earlier versions of Flask-Socketio, however it is unclear why it is still a problem for the version used in the implementation in this paper [6].

C.2.2 Study and Faculty Matching

The current implementation uses less advanced files for recognising studies and faculties due to limitations in the types the backend can give [2]. This also implies that faculty matching is not possible with abbreviations yet.

C.2.3 Chat Switching

As soon as the chatbot starts searching the UvA backend further questions bases on that are not always matched with normal small talk. This usually only happens in cases where the end-user is not seriously searching.

C.2.4 Double Information

Due to double information on the UvA web domain (for example the same page on different study pages) the chatbot tends to return more than one time the same page from a different study. This could be partially solved by saving the page subject, which can already be extracted, to a list of negatively confirmed subjects in order to not return them in the same conversation.

C.2.5 Further Questioning

There are cases in which the chatbot tends to crash internally when a small talk question is asked while he might expect a normal UvA question or a confirmation. See figure 12 for an example.

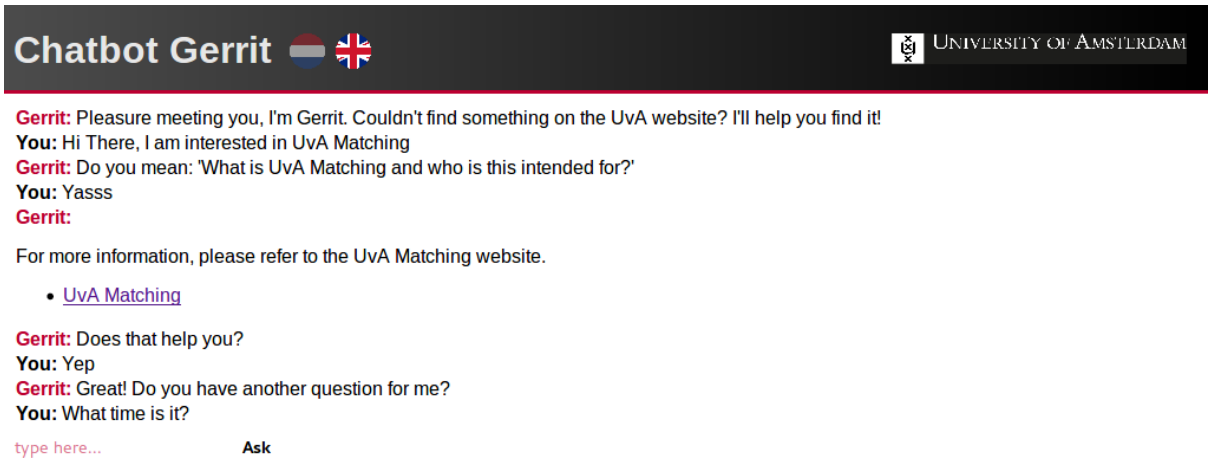


Figure 12: Internal crash on further questioning. Gerrit will not respond anymore.