Christian Paier, BSc

Implementing and Analysing Interactive Recommender Interfaces

Master's Thesis

to achieve the university degree of
Master of Science

submitted to
Graz University of Technology

Supervisor
Assoc. Prof. Dipl.-Ing. Dr.techn. Denis Helic

Co-Supervisor
Dipl-Ing. Lukas Eberhard, Bsc

Institute of Interactive Systems and Data Science

Graz, September 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum

Unterschrift

Abstract

Recommender systems help users to cope with information overload and therefore are important for nowadays web and e-commerce applications. Virtual assistance systems offer a companion for the daily life to fulfill certain tasks. Either in the form of (text-based) chatbots, which can be found all over the web for entertainment purposes or to support users in a more direct way, or voice based virtual assistance systems which are more targeted at command-based tasks.

The aim of this thesis is to connect those concepts and investigate the way users interact with different interfaces for recommender systems.

The first part gives a short overview of recommender system's history and algorithms with focus on collaborative filtering, followed by the chatbot's history and techniques. Finally the history of voice based assistance systems and an explanation of speech recognition is given.

The next chapter describes the implementation of a webform, a chatbot and an Skill for Amazon's Alexa followed by their evaluation with a user study.

Finally the results of the user study discuss that the chatbot was not as well received as the Alexa Skill and the webform, and conclude that this is mainly because the chatbot is not capable of free conversations but was expected to by the users.

The outlook suggests machine learning as one solution for this problem, as it is inevitable for chatbots to support an open conversation to improve the user's experience with them.

Kurzfassung

Empfehlungsdienste unterstützen Benutzer mit einer Flut an Informationen zurecht zu kommen und sind daher wichtig für heutige Webanwendungen oder für den Onlinehandel. Virtuelle Assistenzsysteme bieten dabei einen Begleiter für das tägliche Leben um gewisse vordefinierte Aufgaben zu erfüllen. Dafür gibt es (text-basierende) Chatbots die Unterhaltung oder Kundenbetreuung bieten, oder sprachgesteuerte virtuelle Assistenzsysteme die darauf ausgelegt sind direkte Befehle auszuführen.

Das Ziel dieser Arbeit ist es diese Konzepte zu verknüpfen und die Unterschiede wie verschiedene Interfaces eines Empfehlungsdiensts benutzt werden, zu untersuchen.

Der erste Teil bietet eine kurze Übersicht der Geschichte und Algorithmen von Empfehlungsdiensten, wobei der Fokus auf kollaboratives Filtern liegt, gefolgt von der Geschichte und Techniken von Chatbots. Schlussendlich wird die Geschichte von sprachgesteuerten virtuellen Assistenzsystemen beschrieben, sowie Sprachsteuerung erklärt.

Das nächste Kapitel beschreibt die Umsetzung eines Webformulars, eines Chatbots, und eines Skills für Amazons Alexa gefolgt von einer Evaluierung mittels Benutzerstudie.

Schlussendlich wird diskutiert warum der Chatbot in der Benutzerstudie nicht so gut angenommen wird wie der Skill oder das Formular. Benutzer erwarten eine offene Kommunikation, die der Chatbot nicht bieten kann.

Der Ausblick schlägt als Lösung vor, Techniken des maschinellen Lernens zu verwenden um dieses Problem zu lösen. Es ist unverzichtbar, dass ein Chatbot offen kommunizieren kann, wenn das Erlebnis des Benutzers verbessert werden soll.

Acknowledgements

First I would like to thank my supervisors Denis Helic and Lukas Eberhard for patience, guidance and for reviewing drafts of this thesis.

I also want to thank my parents, Josefa and Anton, my siblings Thomas and Petra for supporting me all those years and encouraging to pursue my studies.

I would especially like to thank my partner and companion Karin for helping, supporting and encouraging me throughout the years of study and discussing late at night many of my (non-)technical questions.

I owe a special thank you to my colleagues at DiCube, especially Thomas who gave me freedom in scheduling work and university but also the chance to develop further when working on many innovate projects.

Contents

List of Figures

1. Introduction

Machines that are capable to communicate with humans in a sophisticated way are subject of many books, the ones able to imitate human are still subject of science fiction. Active research on "intelligent" machines is done since the early days of computing when Alan Turing invented the idea of a test to distinguish "real" from artificial intelligence [92].

Being able to communicate with a machine in an elaborated way changes the way how to interact with it drastically. This leads to a focus shift in computer science not only on the algorithms behind such an artificial intelligence, but also to human-computer-interaction, as the machine's purpose is not based solely on executing lightning-fast calculations any longer.

Those machines are able to cope with the huge amount of information available through the Internet but humans are often left lost and overstrained. Recommender systems are therefore used to help with that. They guide by filtering and ranking news for us, or give recommendations where to eat or what to watch on TV.

Helping humans in nearly every aspect of their life is a reoccurring topic all over science. Virtual assistance systems with speech recognition combine all of the mentioned methods as they are able to communicate in a human-like way and have to be careful when ranking and filtering information as humans are not able to cope with a lot of information at once solely by listening. While such systems have been subject of science fiction literature for a very long time, just think of "HAL 9000" from the movie "2001: A Space Odyssey"[1], they had a major breakthrough with *Apple*[2] releasing *Siri*[3] in 2010.

---

[1] `https://www.imdb.com/title/tt0062622/`

[2] `https://apple.com`

[3] `https://apple.com/ios/siri/`

This master's thesis wants to combine the topics of recommender systems with chatbots and virtual assistance systems by analyzing how users can interact with a recommender system. This thesis is based on a bot that recommends movies to users on the social media platform *Reddit*[4]. The bot and the recommendation framework are part of a project at the *Institute of Interactive Systems and Data Science (ISDS)*[5], at *Graz University of Technology*[6].

The user's interaction is tested with three different interfaces: a classic approach with webforms first, followed by chatbots and voice based intelligent personal assistance is discussed, implemented and analyzed in the next sections.

1.1. Motivation

The success of a recommender system mainly depends on two aspects: the quality of the recommendation itself and the usability of the interface. In the early days user interaction on the Web was not possible at all as the available pages consisted of only static resources. This changed in 2005 when Tim O'Reilly coined the term "Web 2.0" [102] which built a basis for the personal web as it is known today. Nowadays, many web pages are interactive and provide some possibilities for the users to leave their personal mark, from small sections for comments up to big social networks where users can present themselves and connect with each other.

E-commerce web pages were one of the first to implement this personal web approach because their users had to cope with the so-called "Information Overload" [15] where they could not find the right item among the huge amount of other items. Therefore recommendation systems were suggested to provide a set of items that the user might like and therefore to help with this overload of information to reduce work and improve the user's satisfaction [64].

---

[4] `https://reddit.com`

[5] `https://isds.tugraz.at`

[6] `https://tugraz.at`

Nowadays, such systems can be found all over the Internet: *Amazon*[7], *Spotify*[8] and *Netflix*[9] are only three companies who make heavy use of recommendations.

The Web grows to be a more personal place, and it can be see that virtual assistance systems are currently under heavy development. There are mainly two flavours deployed so far: text only software often in combination with instant messaging or chat applications called chatbots, and speech recognition software that gets used either on mobile devices like Siri from Apple or come bundled with special hardware like Amazon's *Echo Dot*[10] devices with the virtual assistant software *Alexa*[11].

The focus of this thesis is to implement and compare three ways of interaction with a recommender system, based on the recommendation framework developed at the ISDS: the classic web page form that can be accessed by every standard browser, implementation described in Section 3.1 the background in Section 2.1, a chatbot provided on *Telegram*[12] and *Facebook*[13] *Messenger*[14]. See Section 2.3 for the theory and Section 3.4 for the implementation and finally a so-called Skill is developed for Amazon's Alexa as one example of a virtual assistant with speech recognition, in Section 2.4 and Section 3.5.

A short introduction of general ways recommendations can be generated is described in Section 2.2.

1.2. Contributions

This project consists of multiple parts of various sources, so we first have to clarify the source of each of those parts. For this thesis a webform

---

[7] `https://amazon.com`

[8] `https://spotify.com`

[9] `https://netflix.com`

[10] `https://www.amazon.com/echodot`

[11] `https://developer.amazon.com/alexa`

[12] `https://telegram.com`

[13] `https://facebook.com`

[14] `https://messenger.com`

(Section 3.1), a chatbot (Section 3.4), an Alexa Skill (Section 3.5), and the API (Section 3.2) that connects them to a database (Section 3.3) are implemented. Additionally, the database storing data specifically for this API (Figure A.2) is created, but the *IMDb*[15] dataset (all movies, persons, genre, for more see Figure A.1) has been provided by the ISDS. A script to preprocess those data (description in Subsection 3.3.1) is written explicitly for this thesis as well. The recommender framework and its bot language, that should be generated by the API, have been provided.

---

[15] `https://imdb.com`

2. Background

This chapter describes the history, theory and algorithms behind each of the implemented parts. We start with the Web's history and its techniques in Section 2.1, followed by a rough overview of recommendation techniques in Section 2.2 Section 2.3 discusses the theory of chatbots and Section 2.4 discusses voice based personal assistant systems.

2.1. Web

Since Tim Berners-Lee proposed and implemented the *Hypertext Transfer Protocol (HTTP)* in 1989 at *CERN*[1], and therefore invented the World Wide Web [52], it has changed radically [36]. The main focus at that time was to exchange and link static information over a network [52]. For this the HTTP method `GET` was defined to fetch a resource from a server via a TCP/IP connection to a host by domain name or IP address. The server was determined to respond with a document of `HTML`[2] formatted text, which gets interpreted by a special application—called "browser" [87]. The first browser, first called "WorldWideWeb", later renamed to Nexus, was developed in 1990, and was able to edit documents as Tim Berners Lee mentions in [89]. The first accessible web page, also from 1990, is still available [86]. In 1994 the *W3C*[3], an organisation to standardize the Web, was founded by Tim-Berners Lee. He is still head of it as of today.

Amongst others, the W3C defines the various `HTML` standards, such as `HTML 5.2`, the most current from December 2017 [50].

---

[1] `https://home.cern/`

[2] Hypertext Markup Language

[3] `https://w3.org`

While in the early days it was sufficient to provide static information, the Web evolved very fast over the years with more active user participation. The first peak was reached with the dotcom bubble starting in the mid-nineties and bursting in 2000. Back then many companies were built that provided services on the Internet for the first time and therefore formed the Web to be a more interactive space [71, 84]. The first major e-commerce companies were founded in those years like Amazon in 1994, or search engines were established like *Google*[4] founded in 1998.

Tim O'Reilly coined the term "Web 2.0" in 2005 [102] to name this new, user centralized and interactive approach where users can generate content by themselves. From a technical perspective *JavaScript* got very popular in that time with the invention of big libraries and frameworks like *jQuery*[5] or *MooTools*[6]. The most important push forward was done with the development of *AJAX*[7] where data could be fetched from a server in the background of a web page without needing a complete reload [42].

With HTML5 in 2014 [49] a new approach for web technologies was standardized by combining HTML with JavaScript and CSS. So not only the HTML elements got refurbished, but also new elements with their own API were defined like the canvas element where one can draw on it using JavaScript.

The latest development of the Web besides HTML5 is the use of full front-end web application frameworks like Google's *AngularJS*[8], or Facebook's *React*[9]. They shift the focus of the Web from not only content providing simple web pages to complete (single-page) web applications. They combine JavaScript with more complex design patterns like model-view-controller (MVC) and are primarily focused on partially re-rendering the shown page for any user interaction.

As part of this thesis, the web application is described in Section 3.1. It provides a form for a recommender system and presents the recommender's

---

[4] `https://google.com`

[5] `https://jquery.com`

[6] `https://mootools.net`

[7] Asynchronous JavaScript and XML

[8] `https://github.com/angular/angular.js`

[9] `https://reactjs.org/`

results. The following section gives a rough overview of recommenders and how recommendations can be generated.

2.2. Recommender Systems

For getting a basic idea of the underlying service the recommender framework from the ISDS provide, a look at what a recommender is and how the techniques needed for such systems is needed. A short and basic introduction into the theoretical concept was done in my bachelor thesis [73], from where the following sections are largely taken and more details can be found.

2.2.1. Overview

There are three main principles for automatically generating recommendations from a set of items [21, 58, 78].

Content-based: Metadata of the items to pick those which are similar to items the user has previously liked are used to recommend items of similar type (e.g., same genre).

Collaborative Filtering: Recommendations are based on finding similar users (or items) based on their ratings. So for any unrated item of a user there might exist a rating from another user from a set of users who had a similar rating behaviour. The idea is that any two users who agreed on certain items in the past are very likely to have similar taste in the future. In reality, however, this assumption is not completely true as users tend to change their taste frequently and quickly. In comparison to content-based recommenders no further knowledge about users or items are needed, besides the given ratings.

Hybrid Recommender: A hybrid recommender mixes those techniques together. They can vary by the combined algorithms but also by the way the results are combined. Those combinations are discussed in [21]. For example, one can add up the results of the various recommenders with weight, or mix the results so that every second item has been generated by a different technique.

Additionally one could use more advanced approaches from machine learning or data science, or combine them with above methods. For example, recommendations can be generated using neural networks, classification, or support vector machines.

All of those techniques use either explicit user feedback, like an upvote, or implicit feedback [78]. For implicit data one has to take the user's behaviour into account, for example, to deduct that a user disliked a movie because it was turned off after watching it for only a few minutes. However, this is nowadays a preferred way to do, mainly because users tend not to give feedback frequently. Collecting implicit feedback means that much more data as basis for the recommendations are available.

2.2.2. Content-Based Recommendations

Using content-based recommendation comes with a big drawback: metadata about all items are needed. As an example one has to record genre, duration, participating actors and cast, language and countries, but also technical specifications like the aspect ratio or color could be useful for a movie recommender. The former engineering director at Netflix stated in 2013 that more than 40 people were needed for hand-tagging their movies [94].

TF-IDF

The term TF-IDF stands for term frequency–inverse document frequency. It is used to determine the importance of a word within a document and to describe a document that way. When using TF-IDF a document gets represented as a vector in an n-dimensional space, where each dimension represents one keyword and consists of two components [58, 78]:

Term Frequency (TF) denotes the relative frequency of a keyword as feature i within a document j, as shown in Equation 2.1. It consists of the absolute value of occurrences of a keyword within the document normalized by the maximum occurrence of all the other keywords within this document.

$$TF(i,j) = \frac{freq(i,j)}{maxOthers(i,j)} \tag{2.1}$$

Inverse Document Frequency (IDF) given in Equation 2.2, measures the importance of a term within the set of all documents. Here N denotes the amount of documents and $n(i)$ the number of documents with this keyword.

$$IDF(i) = \log \frac{N}{n(i)} \tag{2.2}$$

To receive a complete measurement for a word within a document one has to combine TF and IDF as seen in Equation 2.3. This value than can than be used as basis for a recommendation based on the similarity of the TF-IDF values.

$$TF - IDF(i,j) = TF(i,j) \cdot IDF(i) \tag{2.3}$$

2.2.3. Collaborative Filtering

In 1992 the mail system “Tapestry”[44] was presented and showed the principle of collaborative filtering for the first time. It provided a possibility to annotate mails to state whether the mail was liked or disliked by the user and generated—out of this feedback—an individual list of mails. This was used for improving the search of huge amounts of mails.

The idea of Tapestry was expanded in 1994 by the *GroupLens* research group[10] (*University of Minnesota*[11]). They developed a usenet client where filters are automatically generated based on the user’s and other user’s history [77].

When the first big Internet companies were founded in the mid-nineties, recommender systems became adopted widely. Some of those successful companies of that time are:

---

[10] `https://grouplens.org`

[11] `https://umn.edu/`

$$M = \begin{pmatrix} 3. & 1. & 2. & 3. & \cdots & 3. \\ 4. & 0. & 4. & 3. & \cdots & 5. \\ 3. & 3. & 0. & 5. & \cdots & 4. \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1. & 5. & 5. & 2. & \cdots & 1. \end{pmatrix} \tag{2.4}$$

Figure 2.1.: User-Rating Matrix

MovieLens[12]**:** A spin-off of the already mentioned GroupLens project that created the first movie recommender.

Alexa[13]**:** They provided a browser toolbar that analyzed the web pages a user visited and recommended pages that could be of interest. Nowadays the Alexa Ranking still exits, but has been acquired by Amazon.

Firefly: Was the first big music recommender, acquired by *Microsoft*[14] in 1998 and shut down in 1999 [68, 40].

Amazon: Is One of the very big players for recommender systems. They patented item-based collaborative filtering in 1998 [57].

Netflix: Started as a company that remotely rented movies by sending them as DVD to the user. Nowadays they are one of the biggest company for video-on-demand and movie streaming services.
They have been for sure one of the biggest supporter in the research of collaborative filtering, especially since they started the Netflix Prize[15] in 2006 where they offered one million dollar for recommending movies significantly better than the system they had in use at that time. The research group "BellKor's Pragmatic Chaos" won the price in 2009 by building a hybrid recommender [16]. A complete write-up of the solution can be found in [83, 88, 85].

The next sections in this chapter will give a rough overview of algorithms that can be used for collaborative filtering. Especially clustering with k-nearest neighbors (kNN) with Pearson correlation and matrix factorization are discussed. The main difference between those two techniques is that

---

[12] `https://movielens.org`
[13] `https://alexa.com`
[14] `https://microsoft.com`
[15] `https://netflixprize.com`

kNN holds the data needed for the recommendation in memory and is therefore called “memory-based”. For example a user-rating matrix as shown in Figure 2.1 with rows representing a user and columns representing a movie. In contrast, matrix factorization precalculates a model by training with existing data and is therefore called “model-based”.

k-Nearest Neighbours

There are two ways in recommending items when using a neighbourhood-based recommender: either the recommendations are based on finding k similar *users* (user-based neighbourhood) or k similar items (item-based neighbourhood). The first one has the disadvantage that users change their taste quite frequently. Figure 2.2 (taken from [73]) shows the kNN for two movies and nine users. There are multiple users shown together with their ratings for “Movie 1” on the x-axis, and their ratings for “Movie 2” on the y-axis. By using kNN one can spot that the users shown as a circle are closest to the star-shaped user. Recommending movies for a star-shaped user would therefore include all the ratings of the circle shaped users while the ratings coming from the remaining triangle shaped users would not be taken into account.

The disadvantage of k-nearest neighbours is the huge amount of memory that has to be available because the complete matrix of user ratings has to be present in memory during computation. The advantage is the relative easy implementation and the quality of the results. Runtime and quality of the predictions depend mainly on the amount of neighbours k, a higher number of neighbours increases the computation time.

As the kNN algorithm basically consists of calculating a distance between all data points and selecting the k-“nearest” ones for the specific requested data point, one has to decide on a distance on a good measurement technique first. The choice of the measurement does not influence the quality of the prediction significantly [61], so it is sufficient to use an easy or common one, like the Pearson correlation which became a de facto standard for user-based collaborative filtering.

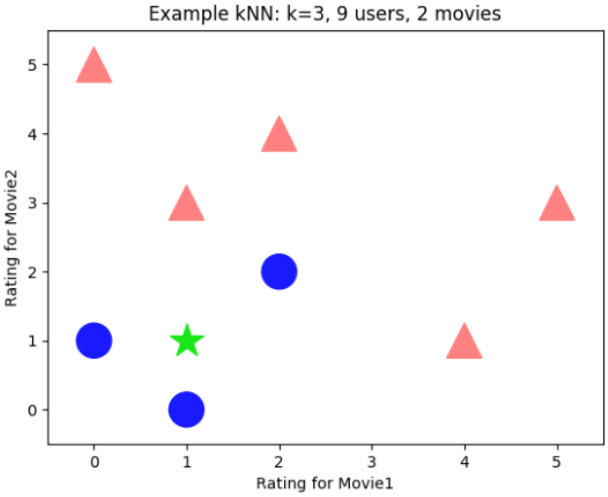


Figure 2.2.: Simple example for kNN with nine users, two movies and $k = 3$ neighbours. The x and y axis are each representing movie ratings, the star, triangles and circles denotes users who rated those movies. All circles are taken into account for the kNN as selected neighbours for the star whereas triangles not. Source: [73]

Pearson Correlation

Equation 2.5 shows how the Pearson correlation can be used to calculate the similarity of two rating vectors a and b by dividing the covariance of the two vectors with the cross product of their standard deviation. a_i, b_i are used for a single rating, and $\bar{a}, \bar{b}$ for the average of the rating vector.

$$sim(a,b) = \frac{cov(a,b)}{\sigma_a \times \sigma_b} = \frac{\sum_i (a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_i (a_i - \bar{a})^2}\sqrt{\sum_i (b_i - \bar{b})^2}} \tag{2.5}$$

Calculate Predictions

Equation 2.6 shows how to calculate the prediction of a rating for one user u combining the user's neighbours which rated that item ($v \in N_i(u)$) and the similarity w_{uv} between them and the user. Here each of the neighbour's ratings gets normalized by subtracting their mean before they get multiplied by the similarity to give more similar users a higher influence. The summation of that for all users divided by the aggregated similarity gives a prediction for a rating, Finally it has to be normalized again by adding the user's mean rating.

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N_i(u)} w_{uv} \cdot (r_{vi} - \bar{r}_v)}{\sum_{v \in N_i(u)} |w_{uv}|} \tag{2.6}$$

Normalizing in both cases is necessary as different users have different views, for example, what is "good" or "excellent": one might hardly give the highest rating, while another user does that frequently. Therefore normalization assures that the ratings are not biased.

Matrix Factorization

A more advanced technique that is be presented here is matrix factorization [58, 78, 104]. The idea is that the user-rating matrix M, with rows representing users and columns representing movies (see Figure 2.1) can be generated by a matrix multiplication $M = UV$. U is a matrix consisting of all user factor vectors u and V is a matrix of all item factor vectors v. A vector u consists of the weight of features — like a movie's genre, actors, or any other metadata — a user might like or dislike, and the item vector v consists of values for features the movie might or might not have, both denoted by a positive or negative number. The problem lies in the fact that many of those features are latent, therefore hidden and often are abstract combination of multiple real-world features (like movie's genres) and cannot be defined manually.

The main approach for matrix factorization, here the UV decomposition, is to find the matrices U and V whose multiplication approximates the given user-rating matrix M.

A typical algorithm finding a good approximation is similar to the algorithm gradient descent:

1. initialize: pick random values for U and V
2. calculate the error of $UV = M'$ by comparing M' to the original M
3. change either U or V

4. repeat steps 2 and 3 until the error gets sufficiently small.

This algorithm gives for any user-rating matrix M with empty fields indicating items that should be recommended, a new matrix $M' = UV$. In general, $M \approx M'$ but the empty items from M are now filled with predictions for a rating.

This above described algorithm suffers from the same problem as gradient decent, as they are very related. Therefore, a good step width has be chosen for changing the values of the matrix. The change should not be too small, as this might never come close to a minimum value, but also not too big to jump over it and overshoot. A good threshold for termination has to be defined, so that the algorithms get a good result, meaning that the error of the approximation $UV = M' \approx M$ is sufficiently small, but does not waste (computation-)time. However there is no guarantee to find the best solution (i.e. to find a global minimum), but for approximation a local minimum should be good enough. Additionally the implementation has to care about overfitting where the error gets very small on the given data, but does not reflect future data, and might therefore lead to wrong recommendations.

General best practices to avoid most of the common pitfalls from above are:

- Execute the algorithm multiple times with random initializations of U and V and calculating the average over all resulting M'.
- Use adaptive step width, to find a minimum very quickly but avoid the danger of over jumping it. Additionally, it decreases computation time by avoiding too small steps.

Calculating the Root Mean Squared Error

A typical measurement for calculating the error is the root mean squared error [78, 73], shown in Equation 2.7, where the square root of the average squares of the given rating r_{ui} and the predicted rating $\hat{r}_{ui}$ is calculated.

$$RMSE = \sqrt{\frac{1}{n} \sum_{u,i \in N} (\hat{r}_{ui} - r_{ui})^2} \tag{2.7}$$

2.2.4. The Big Picture

Putting it all together, a recommender system takes data that represents the user's likeness/dislikeness of a certain item which has to be recorded via an interface. For this thesis three input methods are implemented in Chapter 3. The recommender itself is independent of this interface, waits for any input and generates an output based on it. Whatever recommendation algorithm is implemented, a ranked list of items is generated that ultimately should be presented to the user. It is very common to explain why a specific item was recommended, for example to write "You might like item A because you liked item B in the past". This makes users aware of the way those recommendations work and therefore often increase their satisfaction in the recommended items [58].

As a short overview of the Web is already given, where recommender systems are usually implemented, the following sections will present more unusual interfaces for recommender systems, starting with the history and theory of chatbots, followed by speech recognition based assistance systems.

2.3. Chatbot

2.3.1. Introduction

The Oxford Dictionaries[16] defines "chatbot" as "a computer program designed to simulate conversation with human users, especially over the Internet" [30]. There the origin of this term is dated in the 1990s — however in 1994 Michael Mauldin introduced the term "ChatterBot" for an autonomous player in the multi-user dungeon computer game "TinyMud" [66]. This player is based on ELIZA which is described in Subsection 2.3.2

Being able to talk with machines has been an urge for much longer. In 1950 Alan Turing introduced the Turing test (called the "imitation game"), where an interrogator has to decide the gender of two players A and B by

---

[16] `https://oxforddictionaries.com`

evaluating their responses to the interrogator's questions. Player A has the task to cause the interrogator to falsely identify the players, whereas player B tries to help finding the solution. The main question of the imitation game is the interrogator's performance when player A gets exchanged by a machine. Turing assumed that "in about fifty years' time it will be possible to program computers [. . .] to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning." Further "[. . .] the problem is mainly one of programming". Today's general interpretation of the Turing test is that the interrogators simply determine if the entity they communicate with is a human or a machine [75].

This chapter gives an overview of software programmed to pass the Turing test. However it has to be mentioned that the test itself has been criticised a lot, mainly because only functionality gets tested and not the presence of intelligence (e.g., John Searle in [80]).

Since 1991 the *Loebner Prize*[17] is one real-life application of the Turing test, where the most human-like program gets awarded [48, 63]. This test also recieves a lot of criticism, especially from Marvin Minsky, who stated that the Loebner Prize is an "obnoxious and unproductive annual publicity campaign" [65].

However, Turing more or less laid the foundation for natural language processing (NLP) and artificial intelligence (AI) in general. As NLP consists of mainly two topics: generating and understanding human language, it is the basis for every chatbot. Here mainly the ways a chatbot understands language are discussed, beginning with pattern matching in Subsection 2.3.2 up to more advanced techniques coming from the field of machine learning in Subsection 2.3.6.

This chapter describes the historical background and theory of open domain chatbots that can talk more or less freely, opposed to the chatbot implementation of this master's thesis in Section 3.4, which restricts itself to a conversation that leads to movie recommendations. However it should be clear that the theory behind those are the same, as the difference is for closed domain chatbots the possible restricted in- and output to a certain topic.

---

[17] `http://www.aisb.org.uk/events/loebner-prize`

Additionally it should be mentioned, that the described chatbots mainly communicate in a written form, an extension of chatbots that understands spoken words is discussed in Section 2.4.

2.3.2. Classical Chatbots

ELIZA

In 1966 Joseph Weizenbaum programmed ELIZA, the first program that was able to keep a conversation in natural language [99] and was therefore the first to target the Turing test. ELIZA was implemented to mimic dialogues with a psychotherapist. This approach was chosen to be able to rephrase any input and hide that the program itself has no real intelligence. So ELIZA would respond to the statement "I went for a long boat ride." with "Tell me about boats.", and—as Weizenbaum states—the user would assume that a psychotherapist uses this mechanism to direct the following conversation in a certain direction.

However, ELIZA was said to have passed the Turing test by some people because they really believed that the answers given by ELIZA were human made. As an example a group of psychiatrists saw the program to be the beginning of an automatic form of psychotherapy [98]. An example of a simple conversation with ELIZA is given in Figure 2.3.

Although ELIZA was originally written in the programming language MAD-SLIP [99], an online reimplementation can be found at [34].

PARRY

In 1972 Kenneth Colby reimplemented the ideas of ELIZA and published PARRY, who imitated a person with paranoid schizophrenia [26]. It was considered to be a step forward in regards of ELIZA simply because the chatbot itself seemed to have personality, and could not only rephrase the input sentences, but also changing into new topics, telling small stories and admitting something not to know [66]. It was rather successful as 23 out of 25 psychiatrists considered the bot at that time to be "paranoid" and only

```
U: Hi, how are you?
E: How do you do?  What brings you to see me?
U: My master thesis
E: Why do you say your master thesis?
U: Because I write about chatbots
E: Is it because you write about chatbots that you came to me?
U: yes!
E: I see...  Well, what makes you believe this is so?
```

Figure 2.3.: Conversation of a user (U) with ELIZA (E) (2018-08-17)

two to be "brain-damaged" as they thought that the used language was too simple-minded [26].

A conversation between PARRY and ELIZA can be found at [33].

Pattern Matching

The main technique used for these classic chatbots—which is still popular today—is called pattern matching. Here mainly the input is searched for a keyword (or a sequence of keywords). If such a keyword has been found the corresponding pattern keeps a rule to construct an answer. An example for pattern matching is given by a user saying to ELIZA *"It seems you hate me"*. Assuming ELIZA has a pattern `(0 YOU 0 ME)` this input would be matched, as a the number zero is a place holder for an arbitrary amount of words, for example here between the words YOU and ME and at the beginning. A number greater than zero is a placeholder for a fixed amount of words.

Therefore the input will be split into various parts:

```
(1) It seems that   (2) you   (3) hate   (4) me
```

Assuming the corresponding rule for this pattern is `(WHAT MAKES YOU THINK I 3 YOU)`, the answer ELIZA gives will be `WHAT MAKES YOU THINK I HATE YOU`, because the `3` is a placeholder for the third component of the split input.

In a case where it is possible that multiple pattern match the input it has to be decided which pattern to choose: Any input is parsed from left to right and if a word matches a keyword only those patterns having that keyword (at that position) are taken into consideration [99]. This procedure is not only called pattern matching but can also be described by "stimulus-response" where the pattern represents a stimulus and the corresponding rule as the response and little to no memory of the previous communication [43].

2.3.3. AIML based chatbots

A.L.I.C.E.

Inspired by ELIZA, Richard Wallace released the chatbot A.L.I.C.E.[18] [97, 46, 96]. A.L.I.C.E. still followed the pattern matching approach, however the patterns and transformation rules were implemented by using AIML[19], a XML dialect that was specified by Richard Wallace in 2001 as well.

AIML was intended to be simple and that—if many people write bots with it—these files might get merged to a single so-called "Superbot [4]. An advantage of using AIML is the use of recursive matching with the `<srai>` element and the use of more specialized wildcards instead of ELIZA's numerical placeholders [97]. A detailed insight into AIML is given in [96] the specification of the current version 2 can be found in [7].

A.L.I.C.E. can be described by the terms "case-based reasoning" or "nearest neighbour classification": it generates answers by storing the predefined AIML patterns in a tree and tries to match those with the given input with respect to a given category. Richard Wallace states that A.L.I.C.E. is as simple as ELIZA [4]. However, other than ELIZA, A.L.I.C.E. can recall the last input with the AIML element `<that>`.

Another advantage of A.L.I.C.E. is, that—with the help of the web—10,000 most likely questions were collected, where A.L.I.C.E.'s replies were targeted at.

---

[18] Artificial Linguistic Internet Computer Entity

[19] Artificial Intelligence Markup Language

The chatbot A.L.I.C.E. itself is not directly accessible any more, however it can be used as a library, together with AIML and the successor *Mitsuku*[20] on *Pandorabots*[21] [74].

Mitsuku

Mitsuku is based—like A.L.I.C.E.—on AIML and was written by Steve Worswick in 2012. In contrast to A.L.I.C.E., Mitsuku can reason within specific objects. For example a "tree" can have many properties like "colour: green and brown", or "has: leaves, branches". So if one asks if a tree can be eaten, Mitsuku evaluates the values of "made_from" which has the value "wood", and as "wood" has the property saved that it is not eat able, the bot answers exactly this [54].

An example of a conversation with Mitsuku can be found in Figure 2.4.

Both, A.L.I.C.E. and Mitsuku won the Loebner Prize three times (2000, 2001, 2004 and 2013, 2016, 2017 [48, 63]).

A similar approach, separating patterns and rules within an own language has been done by Bruce Wilcox with "ChatScript"[22], the basis for his chatbots *Suzette*, *Rosette* and *Rose* who won the Loebner Prize four times as well (2010, 2011, 2014 and 2015 [48, 63]).

2.3.4. Jabberwacky/Cleverbot

In 1997 *Jabberwacky*[23], a chatbot implemented by Rollo Carpenter was published on the web [5]. In 2006 Jabberwacky was renamed to *Cleverbot*[24] [24].

Cleverbot generates replies to any input solely by using contextual pattern matching, therefore every unknown sentence is stored in a database, The next time Cleverbot uses this unknown sentence the user's answer is stored

---

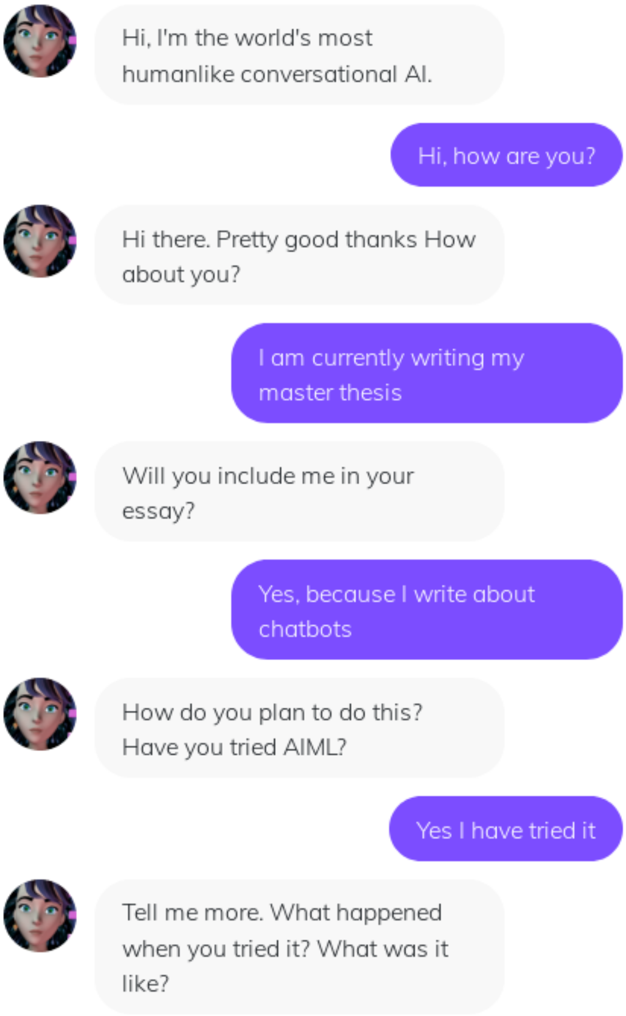
[20] `https://mitsuku`

[21] `https://pandorabots.com`

[22] `https://github.com/bwilcox-1234/ChatScript`

[23] `http://www.jabberwacky.com`

[24] `https://cleverbot.com`



Hi, I'm the world's most humanlike conversational AI.

Hi, how are you?

Hi there. Pretty good thanks How about you?

I am currently writing my master thesis

Will you include me in your essay?

Yes, because I write about chatbots

How do you plan to do this? Have you tried AIML?

Yes I have tried it

Tell me more. What happened when you tried it? What was it like?

Figure 2.4.: Chat example Mitsuku (2018-08-17)

as a possible one. This procedure has one big advantage as there are no hard rules to be implemented, and the bot is even able to "learn" foreign languages [22, 56, 24].

Existor[25], the company that built Cleverbot gives an example for this procedure in [24], beginning with an empty database: The first thing a user will input, will be a simple "Hello". However as Cleverbot does not know this input yet, it is stored in the database. Next, as the chatbot wants to reply, there is only one dataset available, so the answer will be "Hello" as well. Here the user might go on with "how are you?" where Cleverbot learns, that this is a good thing to reply on the initial greeting "Hello" and stores this in the database. In theory, such a bot could store every possible conversation (if storage is infinitely large, and runtime isn't a problem either).

Nevertheless, the quality depends heavily on the available data, for Cleverbot 4-7 million interactions happen per day. To generate a response, 279 million interactions are needed, what they achieve by using large hardware and many optimizations [24].

Even Jabberwacky, which had significantly smaller data available, was able to learn foreign languages although it is not considered to have passed the Turing test, as the responses were often unexpected or senseless.

Jabberwacky won the Loebner Prize twice (2005, 2006 [48]), an example of a conversation with Cleverbot can be found in Figure 2.5.

2.3.5. Chatbots as a Service

With the growth of the Internet, more complex applications moved from installations on local computers to centralized software that gets used and distributed by the Internet. A famous example for this principle, called "Software as a Service" (SaaS), is Microsoft's *Office 365*[26], where applications like *Word*[27] or *Excel*[28] are available online via a monthly paid cloud solution [101].

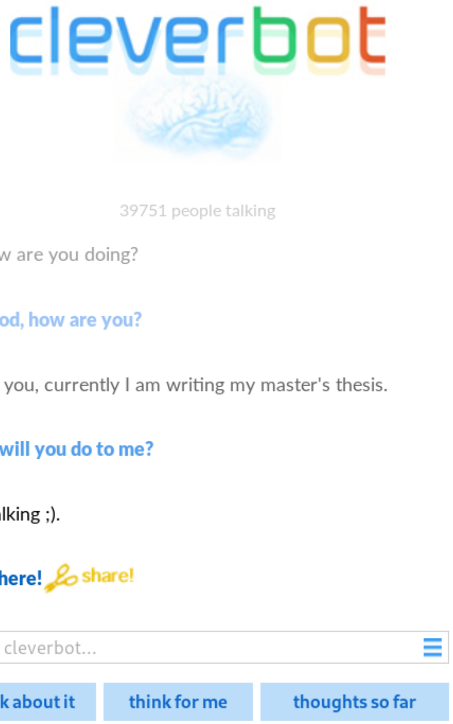
---

[25] `https://existor.com`

[26] `https://office.com`

[27] `https://products.office.com/en/word`

[28] `https://products.office.com/en/excel`



Hi, how are you doing?

I'm good, how are you?

Thank you, currently I am writing my master's thesis.

What will you do to me?

Just talking ;).

Same here!

Figure 2.5.: Chat example between Cleverbot (in blue) and a user (black) (2018-08-23)

Chatbots are no exception of this trend, so nowadays there are many available online services where one can start to build a chatbot, and many of those also support voice input, which will be subject of Section 2.4. A few of those are:

Pandorabots has already been mentioned in Subsection 2.3.3. Offers mainly AIML based chatbot support [74].

Watson[29] IBM[30] built a dedicated computer system (hard- and software!) for artificial intelligence. While Watson (named after IBM's founder Thomas Watson) was not explicitly build for chatbots, one of the main reasons for building it was to create a program that can compete with humans in a real-time *Jeopardy!*[31] quiz game. A requirement was that the system does not use live web search results but stores everything locally [39].

Watson's architecture (called DeepQA) is based on "more than 100 different techniques for analyzing natural language, identifying sources, finding and generating hypotheses, finding and scoring evidence and merging and ranking hypotheses" [39]. So in general Watson interprets the input, executes many parallel algorithms with various interpretations of that input. The results of this executions are than combined and ranked into one final result [39].

The first public appearance was in 2011 (video in [59]), where Watson won a three day Jeopardy! contest against two opponents.

At the end of 2013 IBM published an API that allowed developers to use an even more generalized and extended version—not only restricted to Jeopardy!—of Watson in their own application [51]. As the primary task (generating answers to questions) is rather similar to the principles of a chatbot, there is now the possibility to explicitly generate chatbots. They state to support speech-to-text, natural language classifier amongst other techniques [28].

In 2016 Ashok Goel from *Georgia Institute of Technology*[32] used IBM's Watson as a basis for the chatbot "Jill Watson" for one of his classes. He did not tell the students that Jill Watson was a human but told

---

[29] https://www.ibm.com/watson/

[30] https://ibm.com

[31] https://www.jeopardy.com/

[32] https://gatech.edu

them she was hired as a teaching assistant who is going to answer the students' questions online. However the chatbot was only allowed to answer a question if the system was 97% sure how to answer it. The students did not recognize that Jill Watson was a computer until it was revealed at the final exam [100].

api.ai / Dialogflow[33] api.ai was acquired by Google at the end of 2016 [11] and later renamed to Dialogflow [55]. It runs on *Google Cloud*[34] and includes speech recognition, text-to-speech, and offers various languages and machine learning techniques to improve the natural language understanding of the system [31]. As the basis of the system relies on pattern matching, without machine learning they state that only exact matches for the rules are matched. Additionally to the training data they already include, one can include own training data for the algorithm as well [32]. However Google does not reveal what algorithm or combination of algorithms they use.

The advantage of using those services are mainly due to the simplicity of the offered SDK or API and that they often include multiple languages, big databases and more advanced techniques in the field of machine learning or neural networks. However those techniques use more computation power when the chatbot has to create a response so those services often force the user to the cloud environment of the service provider, which is on one hand an advantage as no separate server has to be acquired, but on the other hand also a disadvantage when thinking of privacy, because communication might include private details like religious beliefs or sexual preferences.

2.3.6. Advanced Techniques

Nowadays chatbots do not consist of only one algorithm but a whole set of algorithms. However most of the new generation chatbot plattforms are closed source so little detail is known what techniques are used or how they are connected to each other. A popular choice is to include additional web search capabilities or databases with huge amount of data to extend the

---

[33] `https://dialogflow.com`

[34] `https://cloud.google.com`

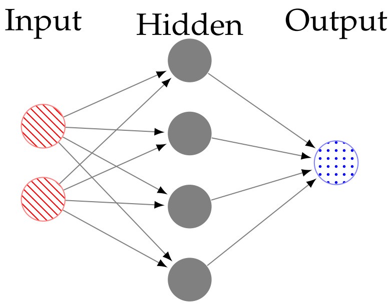


Figure 2.6.: Simple example of a neural network

knowledge base of a chatbot. Like IBM's Watson which uses " a wide range of encyclopedias, dictionaries, thesauri, news articles, literary works" [39].

However, the used algorithms can be any thinkable ones from the field of artificial intelligence, expert systems, machine learning, data mining. This section discusses neural networks as representation of one of those.

Neural Networks

Neural networks are constructed by imitating how the human brain works. They consist of three layers of tiny computational units—called neurons: first the input layer that gets input from outside into the neural network, an output layer that returns the computed values and a finally a layer consisting of a set of hidden neurons who are only used within this network. Each of the used neurons itself has at least in- and output connections and an activation function that determines the output based on the input. Those neurons are highly connected and although each one of them executes only a simple task, due to their connectivity complex calculations can be executed [60]. Figure 2.6 gives a simple visual representation of a neural network, with the input neurons with red stripes, the output neurons with blue dots, and the hidden layers in gray.

Oriol Vinyals and Queoc V.Le (both from Google) describe how to bulild a chatbot with the help of neural networks [95]. They let a neural network map a sequence of words to another sequence and by this they are creating a conversation. In one scenario they used a database with existing chat sessions from an IT help desk, for another scenario movie subtitles. An excerpt of such conversations can be found in Figure 2.7

Another example for the neural networks in chatbots is the parent company of Cleverbot, Existor. They describe their experience building such a chatbot in a post on their company's blog [25], describing a neural network trained with data from previous conversations with Cleverbot. They are optimistic to use neural networks for future implementations. An example of such conversation is shown in Figure 2.8.

2.4. Speech Recognition based Assistance Systems

Virtual assistance systems can be seen to be similar to chatbots: they have to be able to hold conversations but mainly to fulfil a certain task. A very famous example for a virtual assistance system is HAL 9000 from Stanley Kubrick's science fiction movie "2001: A Space Odyssey" from 1968 [35]. Although only represented by a red dot, the computer system HAL is capable of speaking, playing chess and understanding humans. Within the movie HAL is mainly used for steering and maintaining the ship, and also for recreational purposes.

When Apple released Siri in 2010 those science fiction story seemed to have become reality, as it was one of the first (speech based) virtual assistance systems that was available to a broader mass of people.

The challenges of virtual assistance systems and chatbots are nearly the same, the only big difference lies in the voice based input. Subsection 2.4.1 describes how a computer can translate spoken words to a written form. That written form can than be passed on to a chatbot engine as described in Section 2.3. A virtual assistance system normally has to fulfil a certain task, for example to give information about the weather or to create an entry in the user's calendar, therefore the chatbot engine can be considered as

---

[35] `https://www.imdb.com/title/tt0062622/`

```
Machine: hi
Human: hi
Machine: could you please let me know what is
the operating system you are using ?
Human: linux
Machine: ok this is the issue with the system performance , nasser , tools
Human: what is that?
Machine: is it a windows machine?
Machine: we have linux on your machine.
Machine: is it windows xp or windows 7?
Human: i am using linux
[...]
Human: can a cat fly?
Machine: no.
Human: how many legs does a cat have?
Machine: four, i think.
Human: how many legs does a spider have?
Machine: three, i think.
[...]
Human: how much is two plus two?
Machine: four.
Human: how much is ten minus two?
Machine: seventy-two.
```

Figure 2.7.: Examples of conversations between a human and a machine with a neural network based chatbot [95]

```
User: how are you today?
RNN: good.
User: does life have meaning to you?
RNN: sometimes.
User: give me an example.
RNN: example of what?
User: of life the universe or anything.
RNN: me.
```

Figure 2.8.: Example conversation of a neural network based cleverbot [25]

closed-domain because free communication is not possible with most of those systems.

In comparison to chatbots virtual assistants are often controlled by voice input and they can speak by themselves as well. Usually a simple web search is executed to answer specific questions. Many virtual assistance systems come with own devices or are designed specifically for a device and some offeran API to extend the functionality or an SDK for portability to other devices. Examples for such systems are:

Siri[36] Apple was the first one to release a virtual assistant system for the broader audience in 2011 together with the iPhone 4s [90].
The process of developing Siri was initiated by DARPA[37], an American governmental agency to research new technology for the US military and it was developed by SRI International[38] as the largest AI project at that time. A spin-off startup for that technology was acquired by Apple in 2010 [90, 81].
With *SiriKit*[39] Apple released an API for developers to extend Siri with individual features.
Meanwhile Apple has released an designated virtual assistance device, named *HomePod*[40] in 2018 [12].

Alexa[41] Alexa is a virtual assistant explicitly developed for Amazon's Echo and Echo Dot devices and was released in 2014 [9]. Additionally to the chatting functionality Alexa can be used to listen to music, or shop on Amazon or for home automation systems, as due to its popularity many manufacturers are supported [8, 103].
There is an API (Skills Kit) available to extend Alexa's features [19].

Cortana[42] Cortana has been released by Microsoft in 2014 for their phone and desktop operating systems [38].
It has been criticized that Cortana uses Microsoft's search engine,

---

[36] `https://https://www.apple.com/ios/siri/`

[37] Defense Advanced Research Projects Agency `https://darpa.mil`

[38] `https://sri.com`

[39] `https://developer.apple.com/sirikit/`

[40] `https://www.apple.com/homepod/`

[41] `https://developer.amazon.com/alexa`

[42] `https://www.microsoft.com/en-us/cortana/`

Bing[43] and forces links to be opened with *Edge*[44], Microsoft's web browser [69].
To learn a user's behaviour, Cortana stores personal information in a "Notebook" file, where users have control of [70].
Developers can extend Cortana by building own skills for it [27], and with the help of a SDK it can be integrated to other devices or systems as well [53].

Google Assistant[45] Google released Google Assistant in May 2016 as part of *Google Home*[46], their own smart speaker system. It can be seen as a successor of *Google Now*[47], which was able to extend a user's web search with more information, while Google Assistant is said to be able to converse with a user. It can be used either via written text input or spoken words, *Google Search*[48] is used as a search engine [45].
Google Assistant can be extended by using *Actions on Google* [6].

While the basic functionality of the underlying technology is the same for virtual assistance systems as for chatbots as they are often both based on pattern matching that might get combined with machine learning techniques as described in Section 2.3. The main difference is, however, that virtual assistance systems fit seamlessly into human's daily life, especially as they are controllable by voice. This opens a privacy question because speech based virtual assistants have to listen to their surroundings all the time. This has been proven to be a problem in some instances, for example when a privately held conversation was recorded by Amazon's Alexa and sent to another user.[10].

Letting a technical device speak is a rather simple task, the spoken text gets divided into small chunks and for every chunk a certain audio file gets played. For example, Susan Bennet, the voice of Siri, describes this process from a speaker's point of view in an article on cnn.com [1].

---

[43] https://bing.com
[44] https://www.microsoft.com/en-us/windows/microsoft-edge
[45] https://assistant.google.com/
[46] https://store.google.com/product/google_home
[47] https://google.com/search/about/
[48] https://google.com

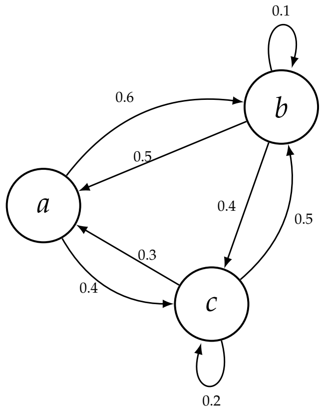


Figure 2.9.: Simple example of a Markov Model

From a technical point of view the more interesting point is how a computer understands voice.

2.4.1. Understanding Voice

This section is based on two different paper on hidden markov models in speech recognition ([41], [76]).

A Markov Model describes a (memory-less) stochastic process in which every possible state depends only on the present state and not on any other in past or future as Equation 2.8 shows.

$$\mathbb{P}(X_{n+1} = x | X_n = x_n, \ldots, X_1 = x_1) = \mathbb{P}(X_{n+1} = x | X_n = x_n) \tag{2.8}$$

Here $X_1, \ldots, X_{n-1}$ describe random variables of states that have occurred in the past, X_n the random variable of the present and X_{n+1} of the future.

It can be drawn as a graph with the vertices representing the states and the edges the transition probabilities between those. The sum of the probabilities of the outgoing edges of one vertex has to be 1. Figure 2.9 is an example for the graphical representation of a simple Markov Model.

To transform voice to text an extension of this model, called Hidden Markov Models (HMM), is often used. The stochastic process in this model hidden and therefore not observable. However the (hidden) state values can be guessed by evaluating other observations. For example, having two unfair coins with uneven probability distribution for the outcome of a coin toss, such a observation could be the result of a coin toss. The hidden states are the information what coin was used for the flip. By evaluating the results it may be possible to estimate which one of the coins was used.

First the audio wave gets split into many tiny fractions and combined into vectors. Those vectors are the observations used in the Hidden Markov Model which should find a sequence of words that corresponds to this vectors, as Equation 2.9 states

$$\hat{w} = \arg\max\{\mathbb{P}(w|Y)\} \tag{2.9}$$

Here $\mathbb{P}(w|Y)$ is the probability that the sequence of words w has been used to generate the observation vector Y (the audio sequence). Using the Bayes rule this can be converted to:

$$\hat{w} = \arg\max\left\{\frac{\mathbb{P}(Y|w)\mathbb{P}(w)}{\mathbb{P}(Y)}\right\} \tag{2.10}$$

Now an acoustic model which consists of various sub-samples (phones)[49] is used to evaluate the likelihood $\mathbb{P}(Y|w)$. Every phone is a HMM where this likelihood is generated and the parameters for this HMM are generated by using training data of existing sets of voice and text inputs. The maximum likelihood is then evaluated to generate a text unit representation of the phones by using constraints of words based on words from a dictionary. If necessary a syntactic analysis is performed to generate meaningful sentences as well.

2.4.2. The Big Picture

In the last two sections chatbots and virtual assistance and the underlying algorithms were discussed. It is obvious that those two concepts related to each other as a virtual assistance can on one hand be seen as an extension of chatbots, as speech recognition and generation and interactive web search

---

[49] The English language consists of 40 phones

are a core part of it, and on the other hand has cutbacks, as chatbots are designed to communicate freely and virtual assistance systems should only fulfill a certain task.

The next chapter (Chapter 3) describes how to implement such a chatbot and how to add an Alexa Skill. It starts with the description of the implementation of a webform, as that will be the basis for comparing the user's behaviour of those interfaces in Chapter 4.

3. Implementation

This chapter describes the implementation of the webform, the chatbot, an Alexa Skill and an API for this thesis. The main focus of this work is to compare various interfaces for a given movie recommender framework and therefore three different concepts are implemented: The implementation of the webform is described in Section 3.1, of the interactive chatbot in Section 3.4 and of the Amazon Alexa Skill in Section 3.5.

These interfaces are evaluated and compared in Chapter 4. The fourth part that is implemented is an API, described in Section 3.2. It handles all the incoming requests from those interfaces and connects them to a database (Section 3.3). An outline of the general architecture and how the various parts are connected is given in Figure 3.1.

The recommender framework itself is part of a project at the Institute of Interactive Systems and Data Science (ISDS), at Graz University of Technology The API implemented for this thesis converts the user's input into a format that the recommender framework can handle. This format has already been developed at the ISDS and is described in Subsection 3.2.2. Any data that can be searched or recommended by using one of the interfaces has been scraped from the *IMDb*[1] by another project for the recommender

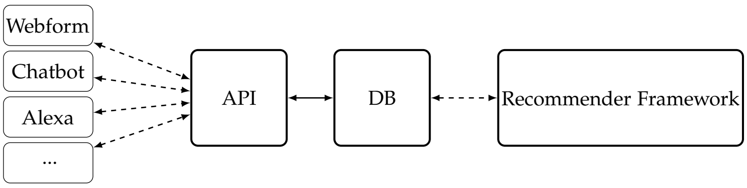


Figure 3.1.: Simplified Architecture

---

[1] https://imdb.com

framework.

The following sections are discussing the details and techniques used for each of the implemented units, beginning by the webform Section 3.1, followed by the description of the API in Section 3.2 and with the database in Section 3.3. Finally, the implementation of the chatbot (Section 3.4) and a Skill for the virtual assistant system Amazon Alexa (Section 3.5) is described.

3.1. Webform

A classic way to exchange information between server and user on the Web is by using a webform. Many web pages have a contact form, where one can input a message that should be sent to the web page owner. However webforms are also used for more interactive user experiences like login forms or shopping carts. Nowadays, as web pages are built with more interactivity in mind, this approach gets replaced by full web applications with additional possibilities for the user [71].

The webform for this thesis is built with such a new approach in mind. Therefore, implementation is done by using Facebook's JavaScript framework *React*[2] combined with the library *Redux*[3] to make manipulating and replaying application states easier. To get any data (e.g., movies, actors,from the server the implemented API is requested via the JavaScript `fetch API`[4] which is used in a Redux middleware (described in Subsection 3.1.2) to send AJAX calls. The detailed routes to the API are described in Section 3.2. Any AJAX call in this section will be shortened for simplicity to the path of the URI, so `http://api.recsys.cpaier.com/v1/genres` will be referred as `/genres`.

---

[2] `https://github.com/facebook/react`

[3] `https://github.com/reactjs/redux`

[4] `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`

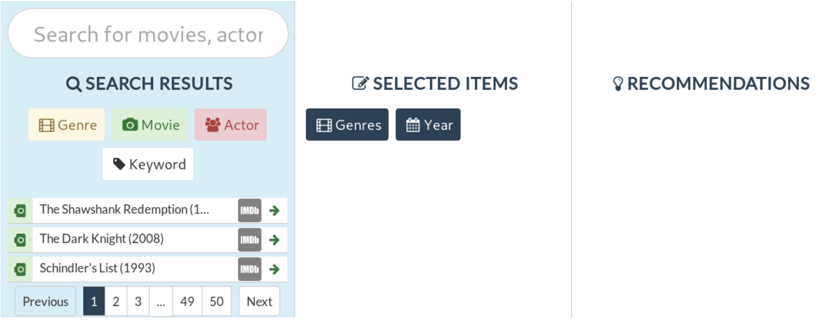


Figure 3.2.: Webform with the default layout of three columns

3.1.1. UI

The visualization of the webform, see Figure 3.2, is organized in three columns, each one with a separate purpose in mind:

- the first third of the screen is used for searching the database
- the second one represents the selected items
- the last one represents the results of the recommendation

When a user accesses the web page a basic set of the most important movies are already displayed on the first section while the other two are empty.

At the very top of the first section a search bar is given. When entering anything an AJAX call is executed that fetches any movie, person or genre from `/items` that fits the input, and updates the displayed item list in the search section. Items are color coded throughout the complete application, therefore genres are displayed with yellow labels, movies are shown in blue, persons in red, keywords in white, and years in blue.

Below the search box there are four buttons (i.e., “Genre”, “Movie”, “Actor” and “Keyword”,) to filter the search results accordingly. Additionally, an item might have one or two buttons: “right arrow”, that selects an item and moves it into the selected items list (second column of the web page), and a button, with the IMDb logo on it, that appears only for movies. It provides a link to the IMDb entry of the movie.

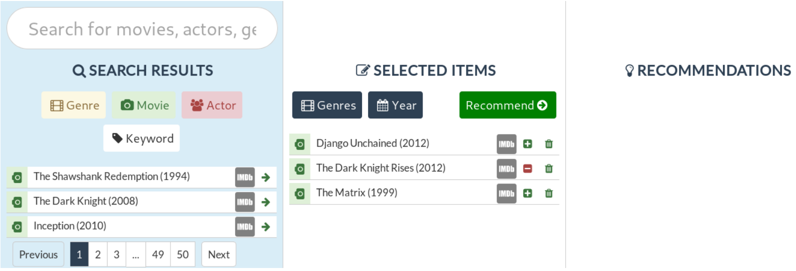


Figure 3.3.: Webform with selected items

At the bottom of the page there is the possibility to navigate through the additional search results. At most there are only twenty search results displayed at once.

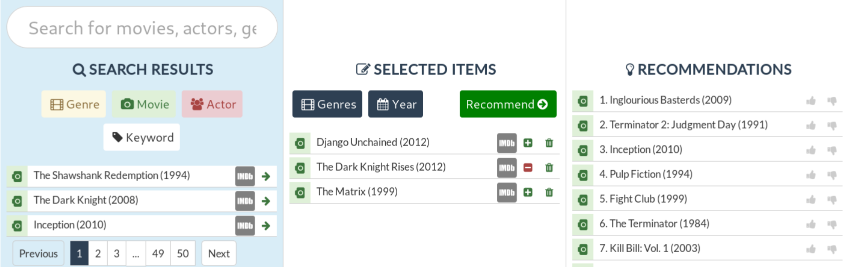
The webform after selecting some items is shown in Figure 3.3.

Each item on the item list can be removed by clicking on the remove icon that is displayed at the end of every item. Additionally, it is possible to set a desired or undesired property which will refine the results the recommender returns on an item basis. Any item with the "minus" sign is undesired by the user and any item with a "plus" sign is desired. In Figure 3.3, the second movie "The Dark Knight Rises" is undesired by the user. Per default any selected item will be seen as a desired one.

Furthermore the second column has buttons to help selecting genres or a year. When moving the mouse over those buttons a popup opens with either a toggle button for each genre, or a range selector to select the range of years the recommended movies were produced. The genres displayed here are fetched from `/genres` on the first load of the application.

To start the recommendation process the user first has to select a few items, then an additional (green) "Recommend" button is displayed on the top of the second column.

When clicking on the button an AJAX call to `/rec/add` is sent to the server to store all the selected items in the database. Then a unique id referring to exactly this set of selected items is generated and returned. This unique id is sent back to the React application which relocates to a URL referencing this recommendation session. An example for such an URL



The Shawshank Redemption (1994)
The Dark Knight (2008)
Inception (2010)

Django Unchained (2012)
The Dark Knight Rises (2012)
The Matrix (1999)

1. Inglourious Basterds (2009)
2. Terminator 2: Judgment Day (1991)
3. Inception (2010)
4. Pulp Fiction (1994)
5. Fight Club (1999)
6. The Terminator (1984)
7. Kill Bill: Vol. 1 (2003)

Figure 3.4.: Webform with recommended items

is `https://recsys.cpaier.com/a4beb63a-2f67-4187-a148-e1f3d0c5a4c6`, with `a4beb63a-2f67-4187-a148-e1f3d0c5a4c6` as the unique id. When a user revisits such a page the occurrence of such an `id` enforces that the set of selected items and recommendation results are automatically fetched from the server if available.

To get recommendation results from the server, the webform tries to poll these from the recommender framework every few seconds (`/rec`) until a valid result is returned. The interval for this polling has to be set to a low value, as a user needs to be informed of the results as soon as possible, but cannot be too low as many AJAX requests will slow down both the user's browser and the server. While waiting for results, any interaction with the selected items and the "Recommend" button are blocked until the results are available.

Just like for the search section, a pagination that enables navigating through all those items that cannot be displayed is provided for the selected items column.

Figure 3.4 shows the webform when the results of the recommendation process are available with the possibility to up- or downvote every recommended movie, where AJAX calls are sent to `/vote/store/up` and `/vote/store/down`. When displaying the results the votes for each movie are fetched by calling `/vote/get` and the corresponding icon is highlighted.

Finally, the recommended movies provide an IMDb link to get more information about them. This URL is wrapped by another URL for improving the results of the recommender, logging what results attracted the user's attention.

To have a complete picture of how a user interacts with the webform every session is recorded. Therefore, a new internal unique session id is generated from the server and fetched by the webform sending a request to `/log/add`. While using the webform every state change (including every action, see Subsection 3.1.2 for a detailed description of React actions and states) gets recorded and stored on the server by sending an AJAX call to `/log/save`. As changes might occur very often and too many of those calls might significantly slow down the user's computer they, are collected before sending them combined with the maximum of every two seconds.

When visiting the `/log` path of the webform a popup opens with the possibility to select a previous recorded session id. After selecting one of those sessions additional controls to replay the recommendation process are shown (see Figure 3.5). The control popup consists of the previously mentioned select box for the session ids and a timeline for a selected session. One can step through the individual state changes with a "next" and "previous" button. Additionally there is a "play" button to automatically walk through the session. The timeline itself shows the position of the current state in respect to every recorded item and can be used to directly address a specific state. Every change of the state in the control popup is directly reflected by the web interface. Above the timeline the timestamp of the currently visible step is show. Below, the completely stored state change is displayed for further inspection.

3.1.2. React and Redux

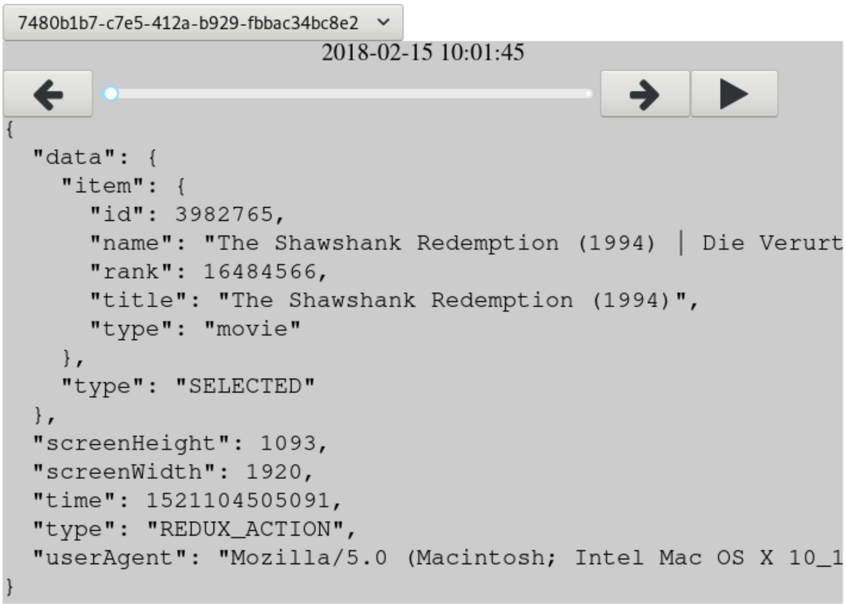
React is one of three popular choices[5] nowadays to create large scaled web applications and is used by big companies like Spotify or Amazon since it was established by Facebook in 2005 [3].

React components are written in JavaScript together with JSX[6], which is a XML-like syntax, to reference a component. A React application consists of

---

[5] others being angular js (`https://angularjs.org/`) and vue.js (`https://vuejs.org`)

[6] JavaScript Syntax eXtension



7480b1b7-c7e5-412a-b929-fbbac34bc8e2

2018-02-15 10:01:45

```
{
  "data": {
    "item": {
      "id": 3982765,
      "name": "The Shawshank Redemption (1994) | Die Verurt
      "rank": 16484566,
      "title": "The Shawshank Redemption (1994)",
      "type": "movie"
    },
    "type": "SELECTED"
  },
  "screenHeight": 1093,
  "screenWidth": 1920,
  "time": 1521104505091,
  "type": "REDUX_ACTION",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_1
}
```

Figure 3.5.: Logging control

various components which all have their own set of immutable properties (called props) and states. Figure 3.6 is a minimal example that shows a button component with the button text as property and a counter that counts all clicks on this button as a state. There are three methods within this component:

- `constructor`: creating the component itself and setting an initial state
- `render`: returns the JSX that should be rendered
- `handleClick`: called by the button's `onClick` event

For `handleClick`, the arrow function syntax, for example `()=>{....}`, is used so that it gets automatically bound to the correct `this` value, which then references the current class as one would expect. `onClick` itself just sets a new state by using the `setState` method. Arrow functions are described by Mozilla in [35] in more detail.

Finally, such a component can be used by importing the button component `import Button from "./Button"` and using the JSX `<Button text="mytext" />` somewhere in the other components render-

```
class Button extends React Component {
  constructor(props) {
    super(props)
    this.state = {
      clicked: 0
    }
  }

  onClick = () => {
    this.setState(prevState => ({
      clicked: prevState.clicked + 1
    }))
  }

  render() {
    return (
      <div>
        <button onClick={this.onClick} text={this.
   props.text} />
      </div>
    )
  }
}
```

Figure 3.6.: React button component

ing function. So it can be seen that properties are passed from outside a component and state is used to handle the component's state.

Handling of all states within an application gets more complicated for bigger projects. Therefore Redux is often used to separate these states from the components and centralizes the access to it. Therefore, it gives the way information flows in the app a fixed structure. Figure 3.7 shows the base structure of a Redux app: the store keeps an (initial) state that is passed to the components and influences their rendering. A component can then call an action, for example, after handling an "onClick" event like in the

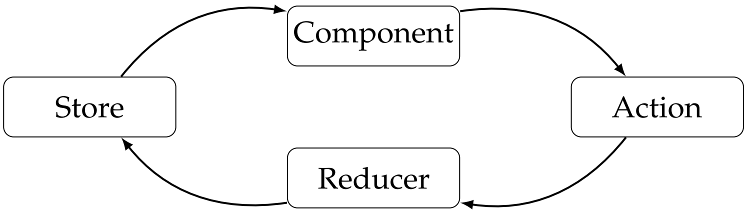


Figure 3.7.: Redux Structure

```
export const count = (data) => ({
  type: "COUNT",
  data
})
```

Figure 3.8.: Redux action example

minimum example Figure 3.6.

The called action gets dispatched and sent to the reducer which updates the state within the store. An action consists of a type and often additional data like in example Figure 3.8. An action count is given that uses the type `COUNT` and takes a parameter `data`. It can be called from a component by using `dispatch(setLoading)`. The function `dispatch` is a special function passed as property. An example reducer is shown in Figure 3.9. If an action with the action type `COUNT` was dispatched, a new state is returned with a different `data` field.

Now it has to be shown how a new state gets passed into a component. Therefore the Redux function `connect` is used to connect a component with the `mapStateToProps` function which sets a `props` field for a wanted state field, as shown in Figure 3.10

Since a browser needs some form of HTML page to render anything the React application has to be combined with HTML code. It needs a basic `index.html` providing some meta data with an empty `<div id="root"></div>` and a script tag to include the React app. For creating this base structure including a tool-chain to merge and minify all the JavaScript one can (and

```
const myReducer = (state = [], action) => {
  switch (action.type) {
    case "COUNT":
      return {
        ...state,
        data: action.data,
      }
    default:
      return state
  }
}
```

Figure 3.9.: Redux reducer example

```
// [..Button Component..]
const mapStateToProps = (state) => ({
  data: state.data,
})

export default connect(
  mapStateToProps,
)(Button)
```

Figure 3.10.: Redux pass data to component example

should) use `create-react-app`[7]. It provides, among other things, those basic files and the commands `yarn start` to run the application in development mode, and `yarn build` to compile, minimize, and optimize the application in production mode[8].

---

[7] `https://github.com/facebook/create-react-app`

[8] More information on Yarn at `https://yarnpkg.com`

```
├── build
├── package.json
├── public
└── src
    ├── actions
    ├── components
    │   └── styles
    ├── constants
    ├── containers
    │   └── styles
    ├── index.js
    ├── middleware
    ├── reducers
    ├── store
    └── styles
```

Figure 3.11.: Webform directory structure

3.1.3. Application Code Structure

The basic directory structure for the webform was created by the previously mentioned `create-react-app` with a modified `src/` directory, shown in Figure 3.11. The main script is `package.json`, which is the main file that `yarn` needs, to know all the dependencies for the application and various commands that shall be provided. `yarn build` writes the output into the `build` directory and the basic HTML file and some publicly accessible files are located in the `public` folder.

The `src` directory is the most important one as it holds all the application specific source code. It contains the folders `action`, `reducers`, and `store` to separate the individual parts for Redux. The components are split into two different folders: (i) `components` which contains React components without access to the application state like simple buttons, headlines, and (ii) the folder `containers` with components having access to the application state. Both folders have a subfolder `styles` for individual component CSS files. Such a directory is also present in the `src` directory itself. In `constants` every constant such as the server URL for the AJAX calls but also the action ids, are stored, and the files in `middleware` are used for background activities, such as handling the AJAX requests to and from the server, and recording and replaying the recorded user log session.

The `index.js` file is the entry point for the application. It configures a new store by using `store/configureStore.js` and starts the playback mechanism (by using `startListeningForPlayback(store)` from the file `middleware/playback.js`), and connects the HTML div element with the id `root` with the applications `Root` component. This component uses `react-router-dom` [9] to switch between the main application in the `App` component and the `Log` component which is displayed when accessing the previously described logging interface (Subsection 3.1.1). Therefore, the `Log` component is just a wrapper of the `App` component and adds the popup and the complete playback logic and mechanism.

The `App` component itself builds the main structure as it includes the `SearchPanel`, `SelectedPanel`, `RecommendationPanel` components which are themselves responsible for one of the three sections of the interface. Additionally, all the error or warning messages are generated in this component.

In the `componentWillMount` method of the `App` component the session id is set. Therefore, `dispatch(getSessionId())` is called, which calls the corresponding `getSessionId` action from one of the files of the `actions` directory. This specific action checks if there is already a session id stored in the `local storage`[10]. If not, a new one is fetched by the server. For fetching such a new session id the `fetchSessionId` function from the `middleware/api` file is used. Finally, as described in Subsection 3.1.2, for both actions a simple action object is dispatched to the reducer which returns a new state.

Here, the action consists of only the `SESSION_ID` constant from the `constants` file as type and the session id, and the reducer therefore solely sets this session id. This state gets inserted as a new `prop` into one of the components in the `container` folder. In the case that the *sessionid* is not present, a default

---

[9] https://reacttraining.com/react-router/web/guides/philosophy

[10] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

value is used by the `defaultProps` function which sets each inserted prop to the corresponding default value from `reducers/defaults.js`.

The same procedure is executed for each property of the components in the `container` folder. For example, when fetching the search results for a specific input the `items` property of the `SearchPanel` component fetches its data like this. In the component itself the `items` property is used in the rendering method as it gets passed to the `ItemList` component, which itself loops through the fetched items list and creates an `Item` component for each of the list entries. The `Item` component is responsible to properly display the data as shown in Subsection 3.1.1. Both the `Item` and the `ItemList` component are in the `components` folder because they do not need direct access to the application state. All the data are solely passed as props.

3.2. API

The API is written in *Python 2.7*[11] and uses the *Flask*[12] framework which is a microframework that can easily be extended but has a robust core that includes basic functionality for web applications [47]. Additionally *mysqlclient*[13] is used to access the database.

The directory structure of the API is shown in Figure 3.12. The main application is located in the `recsys` folder with the files `__init__.py`, `apiv1.py` building the core of the API. These files initialize Flask and provide a function for every incoming HTTP route. `dbase.py` provides access to the database via predefined functions. The file `core.py` only provides some helping functions, and `settings.py` is used to store certain constant values.

`alexa.py` is needed by Amazon Echo which is described in Section 3.5. The files `bot.py` and the subdirectory `brain` in the `recsys` directory are used for the chatbot, as well as `daemon.sh` and `recsys.service`. See Section 3.4

---

[11] `https://python.org`

[12] `https://flask.pocoo.org`

[13] `https://github.com/PyMySQL/mysqlclient-python`

```
├── daemon.sh
├── recsys
│   ├── alexa.py
│   ├── apiv1.py
│   ├── bot.py
│   ├── brain
│   │   └── ...
│   ├── core.py
│   ├── dbase.py
│   ├── __init__.py
│   └── settings.py
├── recsys.service
├── test
│   └── ...
└── wsgi.py
```

Figure 3.12.: API directory

for more information about the chatbot and Subsection 3.2.5 for the description of the daemon for the chatbot interface where the `daemon.sh` and `recsys.service` files are described.

On the same level as the `recsys` folder is the `test` folder which contains unit tests. Currently they are only needed for evaluating the inputs of the chatbot's replies.

Furthermore there is the file `wsgi.py` which is needed as an entry point for the web server. More information can be found in the description in Subsection 3.2.4.

The Python setup is based on virtualenv[14] to create an isolated environment with a fixed setup of Python and Python libraries. Python's corresponding `pip` command[15] is used for installing all the needed requirements which are specified in the `requirements.txt`. Installation can be done by executing `pip install -r requirements.txt`

---

[14] `https://virtualenv.pypa.io`

[15] `https://pip.pypa.io`

```
def create_api():
    app = Flask(__name__)

    app.errorhandler(404)(on_404)
    app.register_blueprint(recsys.apiv1.bp,
   url_prefix='/v1')

    return app
```

Figure 3.13.: Simplified initialization of a Flask App

3.2.1. API Code Structure

The first step when using a Flask application is initialization. A new Flask object has to be created and error handler and blueprints have to be registered. An example of a simplified version of this can be seen in Figure 3.13.

The error handlers are used to react to any invalid request that might raise errors, or exceptions that might be caused by the application itself. Blueprints are used to keep the application more structured by assigning a URL prefix. For this implementation, every request with the URL having the prefix `/v1` gets redirected to the `apiv1.py` file, requests with the prefix `/bot` to the `bot.py` file, and requests containing `/alexa` to the file `alexa.py`.

In addition to the simple initialization a function is registered that gets executed for every incoming request. This function is solely used to create the connection to the database. As this implementation uses two databases (the need for this is described in Section 3.3), there are two connections created and stored within the `flask.g` object, a global parameter that should be used especially for such purposes. So `g.data_db` holds the connection to the IMDb database and `g.api_db` to the applications database.

After this the HTTP CORS header are set. Those are described in Subsection 3.2.3.

Additionally two functions are prepared: `bot` and `bot_cron`. They are annotated by `@app.cli.command` so they can solely be used via the Flask command line interface. The `bot` function provides a possibility to interact

```
[{
  "id":23,
  "type":"genre"
  "name":"Action",
  "title":"Action",
  "link":"",
  "rank":1000000,
},{
  "id":3953016,
  "type":"movie"
  "name":"The Matrix (1999)",
  "title":"The Matrix (1999)",
  "link":"https://imdb.com/title/tt0133093",
  "rank":11100921,
}]
```

Figure 3.14.: ItemList JSON

with the chatbot interface over the command line. The `bot_cron` is needed for a cronjob (see Subsection 3.2.5 for more information).

For the HTTP interface the routes are implemented within the blueprint files. Each possible request to the API expects data to be given either as HTTP GET parameter or as a *JSON*[16] formatted string as the body of a POST request. The returned payload is a valid JSON string. Figure 3.14 shows a formated JSON string of two items in a list, one genre and one movie. With the fields `id` and `type` one can uniquely address any item that could be returned. `id` is the primary key of the database entry and `type` one of `movie`, `genre`, `actor`, `year` or `keyword`. The fields `name`, `title`, `link` are needed for displaying purposes and `rank` is a numeric value defining how important an item is. More important items get a higher rank value and should therefore be displayed above items with a lower rank. The calculation of the rank is discussed in Subsection 3.3.1

An example for the implementation of such a route is given in Figure 3.15.

---

[16] JavaScript Object Notation

```
@bp.route('/genres', methods=["GET"])
def get_genres():
    args = handle_standard_args(request.args)
    return jsonify(fetch_genres(args))
```

Figure 3.15.: Example of the implementation of a basic API route

```
def handle_standard_args(args):
    return {
      "filter": args.get("filter", None, type=str),
      "offset": args.get("offset", 0, type=int),
      "limit": args.get("limit", 0, type=int)
    }
```

Figure 3.16.: Handling standard arguments

Here the parameters of an HTTP GET request, which was sent by a user via one of the interfaces, are handled by the `handle_standard_args` function which can be seen in Figure 3.16. It handles a standardized set of input parameters and is used on multiple occasions. Then the set of request parameters are passed into a database function (here: `fetch_genres`) which executes a *SQL*[17] query. The resulting data might then get handled accordingly and returned to the routing function. This result is then translated into the JSON format using the Flask function `jsonify` which already sets the correct HTTP headers and returns the result to the user.

The following sections describe the various routes of the API within their blueprint.

The `/v1` Blueprint

In Figure 3.17 all the routes with the `v1` prefix are listed together with examples of the incoming and outgoing messages. The route / is used for basic testing and returns the version number. A generic search that is

---

[17] Structured Query Language, a language to query databases

Route	**HTTP**
/	GET
/years	GET
/genres	GET
/items	GET
/rec	GET
/rec/add	POST
/log	GET
/log/add	POST

Figure 3.17.: Base routes on the API

capable of returning every item type is given on the `/items` route, while `/genres`, `/movies` or `/actors` are only returning items with the specific type.

The route `/rec/add` is used to add a new set of selected items to the items list and `/rec` is used to fetch a set of items and recommendation results.

The routes `/log/add`, `/log/save`, `/log/fetch` are needed for recording and replaying the user sessions in the webform. With `/log/add` one creates a new session, `/log/save` is used to add new recorded state changes, and `/log/fetch` is used to fetch the log entries before replaying them. To get a complete collection of available sessions `/log/get` is used.

As the interfaces give the user the opportunity to up- or downvote the recommended movies, the route `/vote/store/up` is used to store upvotes, and `/vote/store/down` to store downvotes. With `/vote/get` the voting rate for all the movies of one recommendation session are requested.

The `/bot` Blueprint

The routes for the chatbot are only used by chatbot interface providers like Facebook or Telegram. As the incoming messages depend on the provider, each of those need at least one separate incoming route to extract the incoming message and user. All currently available routes can be seen in Figure 3.18. At the moment there are only two providers enabled: Facebook

Route	HTTP
/facebook	POST
/facebook	GET
/telegram	POST

Figure 3.18.: Chatbot routes on the API

```
curl -F "url=https://api.recsys.cpaier.com/v1/telegram" \
  https://api.telegram.org/bot<TOKEN>/setWebhook
```

Figure 3.19.: Register webhook for Telegram

Messenger via the `/facebook` route, and Telegram over the `/telegram` route. The Facebook Messenger Platform needs a second route for registering the webhook.

Telegram To create a new Telegram bot one has to talk to the already existing bot @BotFather[18] and initiate the creation with the command `/newbot`. A complete interaction with @BotFather can be found in Figure A.4.

When creating the bot an access token is generated. This token has to be used to register a server URL as webhook where all the incoming messenges are forwarded to. This registration can be done on the command line via curl[19]. An example for that is given in Figure 3.19, where `<TOKEN>` has to be replaced by the previously recieved token.

After registering this webhook the basis communication is finished and incoming messages are already received on the API. An example for such an incoming message is given in Figure A.5. For this thesis only the fields for the user's first name (`message.from.first_name`), the unique chat id (`message.chat.id`) and the inputted text (`message.text`) are used.

As reply one has to send an HTTP POST request to `https://api.telegram.org/bot<TOKEN>/sendMessage` with `chat_id` to ref-

---

[18] `https://telegram.me/botfather`

[19] `https://curl.haxx.se/docs/manpage.html`

erence the same chat, and `text`, the bot's answer as payload of the request. Figure A.6 shows how such an request can be generated with Python.

Facebook Messenger A Facebook Messenger bot can only be used together with a Facebook page. For this thesis I created the temporary CPREC facebook page[20]. For the chatbot itself one has first to register as a Facebook developer by accessing `http://developers.facebook.com` and logging in with an already existing user. Then a new app has to be created on the "Apps" page[21]. These apps can use various products that are prepared by Facebook. Chatbot apps have at least be assigned to the "messsenger" and "webhook" products while creating the app. For the webhook the API URL has to be set. This dialog can be seen in Figure A.7 and can be found in the App builder's "Webhooks" product page.

For the "Messenger" product page one has to generate a new key, assign that webhook and add `pages_messaging` to the used API. All these values are assigned to the Facebook page, an example of this is shown in Figure A.10[22].

A more detailed tutorial for creating a messenger bot can be found in the Facebook Developer Documentation [67]. When the app is configured correctly all messages for this bot are forwarded to the API server. However, until publishing only developers that are assigned to this app are able to communicate with the bot. To make it available for everyone it has to pass a review by Facebook before publishing.

An example message that the API receives from the Facebook Messenger is shown in Figure A.8. For this application only the fields `entry.messaging.message.text` that contains the incoming message, `entry.messaging.sender.id` for referencing the correct user, and `entry.messaging.recipient.id` to be able to send a reply to the correct chat, are used.

---

[20] `https://www.facebook.com/CPREC-174771719834215/`

[21] `https://developers.facebook.com/apps/`

[22] The key is not shown in this screenshot because it gets displayed only once when generated

A reply is done by sending an HTTP POST request back to Facebook, containing the user and the answer. Figure A.9 gives an example for this process in Python. A request is executed to `https://graph.facebook.com/v2.6/me/messages` with the content type "application/json" and a JSON formatted set of the `recipient id` and message. Additionally, `access_token` has to be set to the key hat was generated when creating the Facebook App.

The /alexa Blueprint

The implementation for the Alexa Skills for the Alexa is done with the Flask extension *Flask-Ask*[23], the HTTP routes where an answer should be given (called intent) are described in Section 3.5

3.2.2. Recommender Engine Format

The recommender engine uses its own syntax for handling user inputs. Therefore every input that gets forwarded to the recommender engine has to be translated to this format. Generally, this happens only when the user calls the recommendation engine, for example, by pushing the "Recommend" button in the web interface.

The bot handles any item as a desired one when it is enwrapped with two stars (e.g., `**Titanic**`) and, as undesired, with one star (e.g., `*Titanic*`). The type of an item can explicitly set when prefixing the term with a label and a colon. Following prefixes are available:

- `m:` for movies
- `g:` for genres
- `a:` for any actor or person
- `k:` for keywords
- `y:` for years

---

[23] `https://github.com/johnwheeler/flask-ask`

```
cors = CORS(app, resources={r'/*': {"origins": '*'}})
```

Figure 3.20.: Using flask-cors to enable cross-domain AJAX requests

So `*m:Titanic*` will explicitly for the bot to handle “Titanic” as an undesired movie.

When adding a year constraint one can use > to state that movies later than a certain year should be recommended and < to include years earlier than the given one. So `y:>2008` would include the years after 2008.

If no type is given, the engine tries to guess the type by matching it in a fixed order with the corresponding data: genres, movies, actors and keywords. So the recommender engine tries to find a genre with that same input, and if there are no results the movie titles are evaluated, and so on.

3.2.3. CORS

Cross-origin resource sharing (CORS) is a standard by the W3C and *WhatWG*[24] that specifies the behaviour of web resources when requested from a different domain as the initial one. For example, AJAX requests are blocked by that mechanism when sending a request from domain A to B. However, this API should be accessible by different providers, such as Telegram or Facebook and therefore, it is necessary to disable CORS. This can be done by setting specific HTTP header. For Flask, the extension `flask-cors` provides possibilities to control that in a simple way. An example implementation in enabling AJAX cross domains is given in Figure 3.20.

---

[24] `https://fetch.spec.whatwg.org/`

3.2.4. Server

For the live setup *Nginx*[25] as server software together with *uWSGI*[26], and SSL certificates provided by *Let's Encrypt*[27] to access the Python application via the Internet, are used. An example configuration for uWSGI can be found in Figure A.13, for Nginx in Figure A.14.

3.2.5. Daemon

For the chatbot implementation a cronjob-like daemon is needed that checks the database every second for a new recommendation result, and updates the corresponding chat. A simple Bash script is used that calls the `bot_cron` function implemented within the Flask API as additional command line interface. This Bash script can be found in the API root folder as `daemon.sh`. To ensure that this script gets started when the system boots, and restarted when it may crash, the system and service manager *systemd*[28] is used. Therefore, the file `recsys.service` has to be in a folder accessible by systemd, for example `/etc/systemd/system`.

3.3. Database

The relational database system *MySQL*[29] is used, which is not only a common choice but also because the already existing recommendation uses one.

The implementation uses two different databases: one for data from the IMDb that were created by another project at the Institute of Interactive Systems and Data Science (ISDS), at Graz University of Technology. The most important tables this database are explained further below. The other

---

[25] `https://nginx.com/`

[26] `https://uwsgi-docs.readthedocs.io/en/latest/`

[27] `https://letsencrypt.org/`

[28] `https://freedesktop.org/wiki/Software/systemd/`

[29] `https://www.mysql.com`

database has been created solely for this application and is used by the API to store data from any of the interfaces, and the results from the recommendation process itself.

A detailed outline of the used tables and columns for the `imdb` database can be seen in Figure A.1, and for the `api` database in Figure A.2 and Figure A.3.

The `imdb` database is merely used for the movies and data that were scraped from the website *IMDb.com*. Mainly, the following tables are used:

- `genre`: keeps all the genre data
- `person`: provides every actor, director, or other person in any of the movies
- `movie_person`: connects those persons with the movies
- `movie__matched`: containing the data for movies with more than 1000 votes
- `movie__matched_alternative_title`: containing alternative titles for movies

There are currently[30] exactly 31 genres, 5.224.003 persons, and 11.578 movies in those tables.

The `api` database is the core of this implementation and consists of the tables:

- `recommendations`: stores all the selected items. The recommendation process takes those items and stores its results in the `recommendation` field of this table
- `log`: used for recording the user's session from the webform
- `bot`: used for recording the user's session when using a chatbot
- `person`: extends the person's data from the `imdb` database by a rank
- `alternative_genres`: extends the genres from the `imdb` database with alternative names
- `vote_log`: stores every single up- or downvote for a recommended movie together with the IP address and session id
- `votes`: the cumulative amount of votes for every movie

---

[30] 2018-04-03 02:26

```
SELECT
  FLOOR(
    COALESCE(m.rating_votes, 0)*
    COALESCE(m.rating_rank, 0)
  ) as rank
FROM movie__matched m
```

Figure 3.21.: Calculating the rank of a movie (SQL)

3.3.1. Preprocessing the Data: Ranking

The interfaces need to display persons, genres and movies in one consecutive list, so a ranking has to be defined where specific items are placed. For genres, this task is rather simple: there are currently only 31 different genres that should be placed at the very top of the list. Therefore, they get assigned a very high constant rank value of 1,000,000.

For movies, the rank considers the amount of ratings with their total rating value. A movie with very high rating value but only few ratings might have to be ranked lower than one where the average rating is slightly lower but the amount of ratings is very high. Hence, the actual rank is a multiplication of the fields "rating_votes" with "rating_rank". This calculation can be done ad-hoc for each request as it does not decrease the time such a query needs for execution significantly. Figure 3.21 shows how the rank of a movie can be calculated by using a SQL query.

For ranking persons in a rudimentary way the average ranking of all the movies they have participated is considered. This simple approach however comes with a big flaw as those persons who participated in only one movie which was rated with a very good rating will be pushed to a higher rank compared to an actor with many assigned movies. Therefore, a normalization factor is used. The equation for the rank of a person can be seen in Figure 3.22, where n is the amount of movies assigned to this person, $votes$ are the total amount of votes, and $vrank$ is the average vote (of the range of 1-10 from the IMDb) of one movie.

For this factor, an average person is assumed to have been part of the cast

$$rank = \left(\frac{1}{n} \sum_{k=1}^{n} votes_n * vrank_n \right) \cdot \frac{n}{50}$$

Figure 3.22.: Calculating the rank of a person

```
SELECT FLOOR(
  AVG(
    COALESCE(m.rating_votes, 0) *
    COALESCE(m.rating_rank, 0)
  ) * (COUNT(1)/50)) as rank
FROM movie_person mp,
  person p, movie__matched m
```

Figure 3.23.: Calculating the rank of a person (SQL)

or crew in fifty movies. Now the before mentioned average of their movie ratings gets multiplied by the amount of movies they participated divided by this normalization factor. So a person with less than the assumed average movies assigned gets a lower ranking, but one with much more movies gets pushed a little bit.

An implementation in SQL can be seen in Figure 3.23. As this calculation is very time-consuming this data needs to be preprocessed by calculating the rank for every actor in the `imdb.person` table with movies from `imdb.movie__matched`. The result gets stored in the `api.person` table.

3.4. Chatbot

The chatbot is implemented with *RiveScript*[31]. Python is used as RiveScript interpreter[32] because the logic of the bot can be combined within the API to keep the interfaces for the bots separated from the logic.

---

[31] https://www.rivescript.com/

[32] https://github.com/aichaos/rivescript-python

```
+ my name is *
- Hi <star>!
```

Figure 3.24.: Simple RiveScript example

The multiple bot interfaces and their individual routes are handled by the Flask framework which handles the different message formats and restructures them to one common base. This gets passed into the logic component of the implementation which is the same for all the bot service provider.

3.4.1. RiveScript

Language Overview

RiveScript is a scripting language supporting various languages like *Go*[33] and Python. The advantage of RiveScript in comparison to its competitors is that the language is dynamic and easy to learn as, for example, AIML[34] which uses XML, that gets rather complicated and not as dynamic for bigger projects. Another advantage is that no additional running service is needed (like for ChatScript[35] [23]).

An in-depth reference can be found in [79]. The syntax is simple and mainly consists of stating input and output messages. One example can be found in Figure 3.24, where it can be seen that any input (marked with the "+" sign) that starts with "my name is" gets matched as "*" which denotes a wild card. As `<star>` gets replaced by whatever was matched in the previous line, the bot would answer to the message "my name is joe" with the reply "Hi joe!".

Of course the language offers more possibilities than this example: such as surrounding parts by square brackets defining optional statements, so

---

[33] `https://golang.org`

[34] `https://www.pandorabots.com/docs/aiml/aiml-basics.html`

[35] `https://github.com/bwilcox-1234/ChatScript`

```
+ do you like (cars|bicycles) {weight=20}
- Yes!

+ [*] like (cars|bicycles)    {weight=10}
- Ok.
```

Figure 3.25.: Advanced RiveScript example

a statement “I like [pink] elephants” would match both “I like pink elephants” and “I like elephants”. When having multiple choices that should be matched one can use parentheses with a vertical bar to separate the various choices. So “I like (ice cream|broccoli)” would match “I like ice cream” and “I like broccoli”.

A more advanced example of RiveScript is shown in Figure 3.25. There are two different sets of in- and outputs. For both input statements a weight is set. The first has assigned a weight of twenty, the second of ten. A statement with a higher weight gets evaluated before a lower one. So the question “Do you like cars?” would be matched by the first statement and the bot answers with “Yes!” and never be matched by the second statement.

The file `begin.rive` is the entry point of the interpreter and should be used to handle global variables. Arrays of words, substitutions of words and other functions could be set. for the RiveScript interpreter. In Figure 3.26, one basic example for a begin file is given. The version for the RiveScript interpreter is set and the debug flag is set to true, so that one can investigate how every input gets matched. In productive setups it is set to `false`. Then a word substitution, which replaces the input *i'm* to *i am*, is defined by using the keyword `sub`, and finally an array of words is defined. Such an array can be used for synonyms, so that one can use `@like` instead of the before mentioned syntax from the advanced example (Figure 3.25). If a line starts with ˆ it is a concatenation of the line before. One can define that the lines should be concatenated with a newline by using the command `! local concat = newline`. Alternative values are `space` where lines are concatenated by adding a simple space character or `none` where nothing gets added.

```
! version = 2.0
! global debug = true
! sub i'm = i am
! array like = like|love|enjoy
^ desire|want
```

Figure 3.26.: RiveScript begin.rive example

```
> object sum python
  return int(args[0]) + int(args[1])
< object

+ what is # + #
- <call>sum <star1> <star2></call>
```

Figure 3.27.: RiveScript object example

As RiveScript's purpose is to give the opportunity to parse incoming messages, match them to a set of rules, and create an answer message, it is not capable of executing more complex calculations. For this purpose, one has to use object macros which are written in a full programming language like Python or *Perl*[36]. Figure 3.27 gives an example for an object call of a Python function `sum` which adds two numbers. The two numbers are extracted from the incoming message by using `<star1>` and `<star2>`, and passed into the object function as argument vector.

To make implementation easier, object calls can be excluded in a Python file. This function gets two parameters: the RiveScript interpreter instance, and the argument vector. Additionally, one has to call `set_soubroutine` for each of the functions to register them beforehand when starting RiveScript from Python (Subsection 3.4.2 for more information).

---

[36] `https://www.perl.org/`

```python
bot = RiveScript(utf8=True)
bot.unicode_punctuation = re.compile(r'[!?.:]')
bot.load_directory("./recsys/brain")

def search_movies(rs, args):
    # [...]

def get_reply(user, message):
    return bot.reply(str(user), message)

def init(user, username, recipient, provider, style):
    # [...]

bot.set_subroutine("movie", search_movies)
# [...]
```

Figure 3.28.: Extract of starting RiveScript

3.4.2. Code Structure

The listing of the necessary files and the brain directory for this implementation can be seen in Figure A.11. The main center is the `bot.py` file where the incoming HTTP requests are handled as described in Subsubsection 3.2.1. Additionally, RiveScript gets initialized there. The basic lines for this initialization from the actual implementation can be seen in Figure 3.28.

There RiveScript itself gets initialized and set which punctuation characters should be ignored when evaluating an incoming message. Then the `brain` directory is loaded where all the RiveScript files are located. For any incoming message the `init` method is called, and sets variables in RiveScript by using `set_uservar` if not already present from a previous session. As indicator to separate the various incoming, user a `user id` or a `chat id` from the provider is used. RiveScript keeps track of the following information for every user: the user's name and its id to address them directly, a recipient and provider to be be able to write responses via the cronjob, the amount of items that should be displayed at once, and the style since different interfaces allow different ways of formatting messages.

As last step, the `replay` method has to be called with the user's identification and its incoming message. This causes RiveScript to start the matching process for the best fitting answer. If an object call is made, this call gets redirected back to the `bot.py` file, since all these methods are implemented there and registered by using `set_subroutine`. This registration is shown above by the example `search_movies`.

The RiveScript brain files for the interpreter are structured in a way that each file has a specific atomic purpose. So the file `year.rive` handles all the inputs with one of the possible "year" inputs.

Each file consists of a collection of rules for this purpose where each rule has an assigned weight consisting of 4 digits. This weight ensures the correct order of matching. The first digit denotes the group the rule belongs to (for example every rule in the "year" file has to be between 8000 and 8999).

This ensures that the statements get matched exactly in the order as described in Figure 3.29

pos	weight	file	items
1	9000-9999	base.rive	base commands
2	9500-9699	select.rive	select, deselect
3	9000-9700	help.rive	help
4	8000-8999	actor.rive	actors
5	6000-6999	keyword.rive	keywords
6	5000-5999	actor.rive	actors
7	4000-4999	genre.rive	genres
8	3000-3999	vote.rive	vote
9	2000-2999	movie.rive	movie
10	1000-1999	item.rive	item

Figure 3.29.: The order in which RiveScript evaluates the rules

The second digit differs between negative (1) and positive (0) statements. It is important that negative statements are matched first, otherwise, for example, sentences with "not like" would be matched by the wrong category when using rules similar to "[*] like". The rest of the digits is to keep a certain order of rules within the same file.

Each of the rules is mapped to one base command (for example `likemovie *`) which is just a wrapper for the object calls to a Python function. This function handles the connection to the database for this input. For example, it searches for movies for the `likemovie` function or stores the selected results when calling the `recommend` function. The base commands are implemented in the `brain/functions.rive` file and the implementation of the object calls in `bot.py`. A complete list of all available base commands and object calls is given in Figure 3.30. Any object call that searches for an item receives a plus (+) or a minus (-) sign to denote if an item is desired or undesired by the user. Every object call gets a number for debugging purposes assigned to identify which rule matched the input sequence. For any of the commands this number gets passed the list of matched words from the input as `<star>` tag.

Finally, any of the bot's answers is either placed directly in the brain files, or generated by the object calls.

A full interaction with the chatbot can be found in Figure 3.31 and Figure 3.32.

3.5. Alexa Skill

Alexa Skills are tiny applications written explicitly for Amazon's voice service Alexa. A detailed reference can be found at the online Amazon Alexa Documentation [20]. To create a skill it is necessary to register and login via the Amazon developer web page[37] and to create a new empty skill at the *Alexa Skill Console*[38].

A skill consists of an *Invocation Name* and a collection of *Intents* with *Utterances* and *Slots*. The `Invocation Name` should consist of more than one word, as it is easier for Alexa to recognize it, and is needed to refer the input

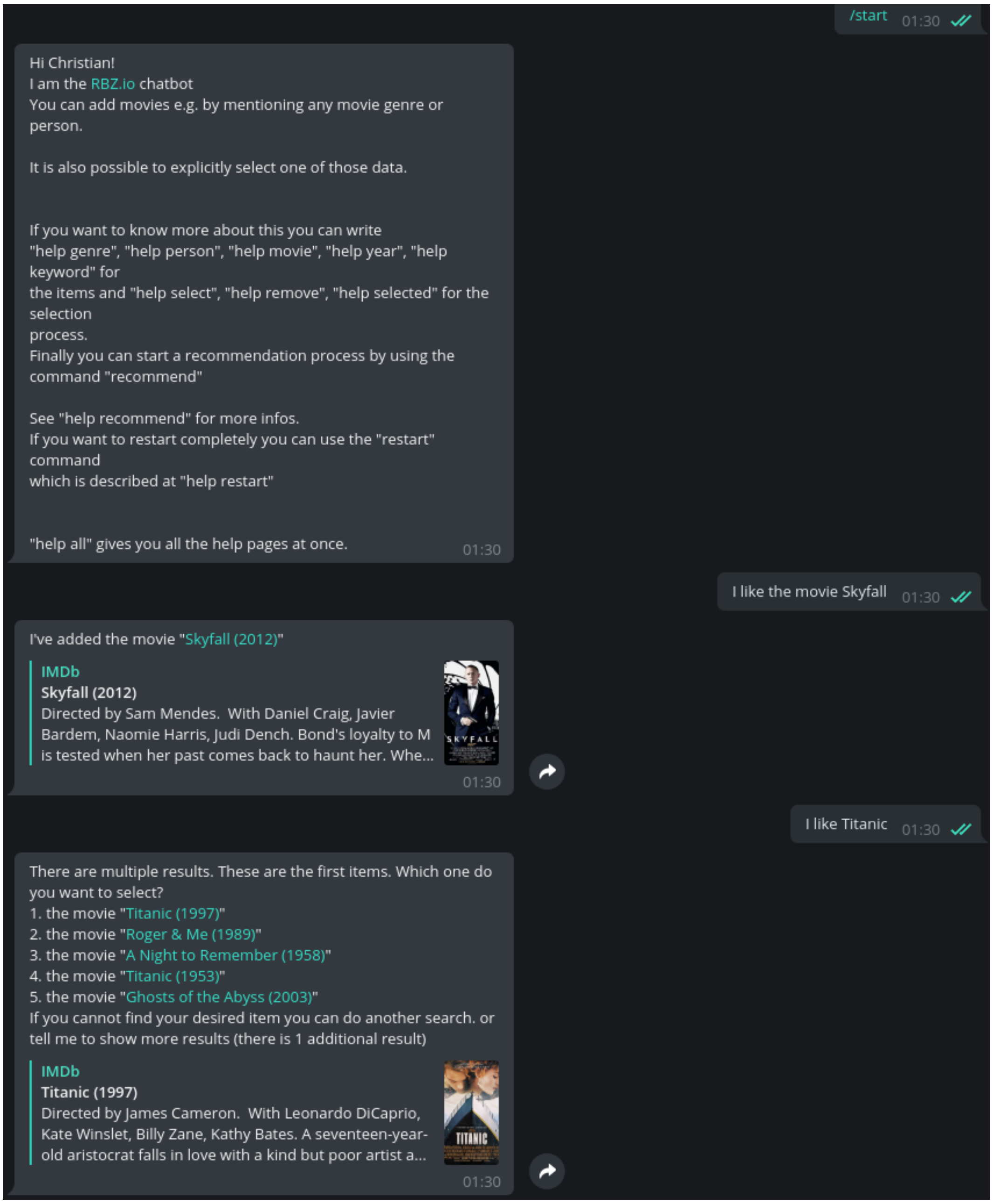
---

[37] https://developer.amazon.com

[38] https://developer.amazon.com/alexa/console/ask/

command	**object call**	**purpose**
`[*] dislikegenre *`	genres	selects an undesired genre
`[*] likegenre *`	genres	selects a desired genre
`[*] dislikeactor *`	actor	selects an undesired actor
`[*] likeactor *`	actor	selects a desired actor
`[*] dislikeyear *`	year	selects an undesired year
`[*] likeyear *`	year	selects a desired year
`[*] dislikekeyword *`	keyword	selects an undesired keyword
`[*] likekeyword *`	keyword	selects a desired keyword
`[*] dislikemovie *`	movie	selects an undesired movie
`[*] likemovie *`	movie	selects a desired movie
`[*] dislikeitem *`	dislikeitem	selects an undesired item
`[*] likeitem *`	likeitem	selects a desired item
`show @select *`	select	select item from list
`show @selected *`	selected	show selected items
`recommend *`	recommend	start the recommendation process
`pageup *`	pageup	call the next page
`pagedown *`	pagedown	call the previous page
`upvotemovie *`	upvote	upvote a movie
`downvotemovie *`	downvote	downvote a movie

Figure 3.30.: Commands and object calls for the chatbot



/start 01:30

Hi Christian!
I am the RBZ.io chatbot
You can add movies e.g. by mentioning any movie genre or person.

It is also possible to explicitly select one of those data.

If you want to know more about this you can write
"help genre", "help person", "help movie", "help year", "help keyword" for
the items and "help select", "help remove", "help selected" for the selection
process.
Finally you can start a recommendation process by using the command "recommend"

See "help recommend" for more infos.
If you want to restart completely you can use the "restart" command
which is described at "help restart"

"help all" gives you all the help pages at once. 01:30

I like the movie Skyfall 01:30

I've added the movie "Skyfall (2012)"

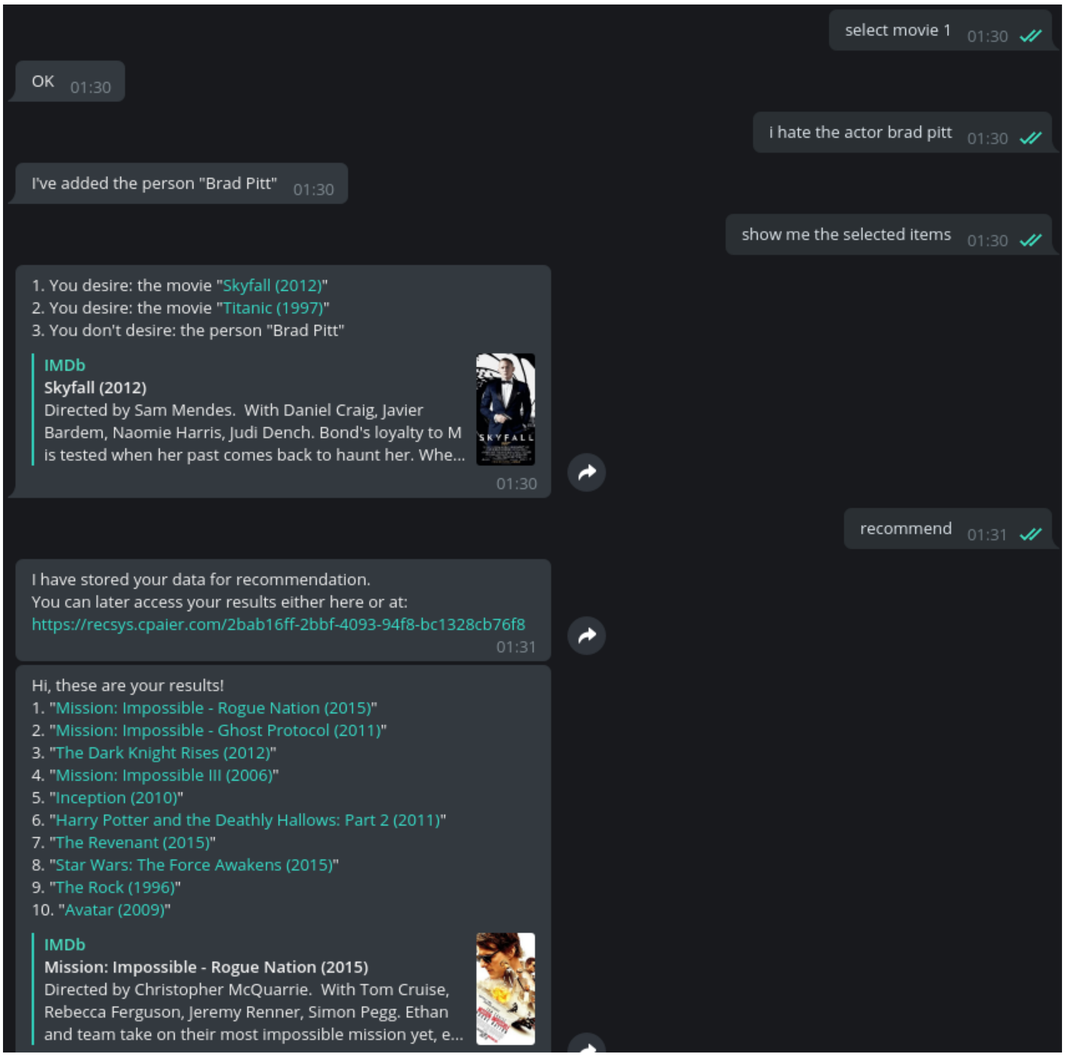
IMDb
Skyfall (2012)
Directed by Sam Mendes. With Daniel Craig, Javier Bardem, Naomie Harris, Judi Dench. Bond's loyalty to M is tested when her past comes back to haunt her. Whe... 01:30

I like Titanic 01:30

There are multiple results. These are the first items. Which one do you want to select?
1. the movie "Titanic (1997)"
2. the movie "Roger & Me (1989)"
3. the movie "A Night to Remember (1958)"
4. the movie "Titanic (1953)"
5. the movie "Ghosts of the Abyss (2003)"
If you cannot find your desired item you can do another search. or tell me to show more results (there is 1 additional result)

IMDb
Titanic (1997)
Directed by James Cameron. With Leonardo DiCaprio, Kate Winslet, Billy Zane, Kathy Bates. A seventeen-year-old aristocrat falls in love with a kind but poor artist a... 01:30

Figure 3.31.: Example Telegram Chat Page 1



select movie 1 01:30

OK 01:30

i hate the actor brad pitt 01:30

I've added the person "Brad Pitt" 01:30

show me the selected items 01:30

1. You desire: the movie "Skyfall (2012)"
2. You desire: the movie "Titanic (1997)"
3. You don't desire: the person "Brad Pitt"

IMDb
Skyfall (2012)
Directed by Sam Mendes. With Daniel Craig, Javier Bardem, Naomie Harris, Judi Dench. Bond's loyalty to M is tested when her past comes back to haunt her. Whe...
01:30

recommend 01:31

I have stored your data for recommendation.
You can later access your results either here or at:
https://recsys.cpaier.com/2bab16ff-2bbf-4093-94f8-bc1328cb76f8
01:31

Hi, these are your results!
1. "Mission: Impossible - Rogue Nation (2015)"
2. "Mission: Impossible - Ghost Protocol (2011)"
3. "The Dark Knight Rises (2012)"
4. "Mission: Impossible III (2006)"
5. "Inception (2010)"
6. "Harry Potter and the Deathly Hallows: Part 2 (2011)"
7. "The Revenant (2015)"
8. "Star Wars: The Force Awakens (2015)"
9. "The Rock (1996)"
10. "Avatar (2009)"

IMDb
Mission: Impossible - Rogue Nation (2015)
Directed by Christopher McQuarrie. With Tom Cruise, Rebecca Ferguson, Jeremy Renner, Simon Pegg. Ethan and team take on their most impossible mission yet, e...

Figure 3.32.: Example Telegram Chat Page 2

to the correct Skill application. Setting `movie recommender` as an invocation name means that sentences similar to "Alexa, tell the movie recommender that I like Inception" will match to open this skill. The part of the sentence after the invocation name will be handled by the skill itself. Intents are used to match the second part of the sentence. An Intent consists of a unique name and at least one sample Utterances. An Utterance is an sample of input that should be matched. For the above example the utterance "I like Inception" will match the input.

To match "I like Forrest Gump" with the same intent it is preferred to mark the movie title as Slot. Slots are marked by surrounded braces ({}). An example for an utterance with a slot would be "I like {Movie}". However Alexa needs a type dedicated to each slot. Amazon delivers many predefined and very specialized types like `AMAZON.AT_CITY` which will match every city in Austria, but also custom types are possible. For this implementation most of the slots have the type `AMAZON.SearchQuery` as this setting will guarantee that every input will be matched, but also `AMAZON.FOUR_DIGIT_NUMBER` is used for year numbers and `AMAZON.NUMBER` for numbers in general. The last one is especially important as Alexa otherwise might interpret the word "three" as "free", which is — of course — not a number. The interface for creating an intent, its utterances and its slots on the Amazon Alexa Console can be seen in Figure A.12.

The complete configuration with all intents, utterances, slots and the invocation name can also be uploaded (or downloaded) as JSON file. A short example for this file is given in Figure 3.33. There, the invocation name is set to `movie recommender` and a sample intent `LikeMovieIntent` is set with one slot (`Movie`) with the type `AMAZON.SearchQuery`. Additionally, there are two samples given for this intent.

To connect the Alexa Skill to a server one sets the link in the "Endpoint" section of the Builder[39]. Please note that only HTTPS is supported and the application has to be prepared to receive JSON POST requests.

When every intent has been set up correctly, pushing the "Build Model" button generates the Skill for usage. After building it, the Skill can be tested,

---

[39] Only Web services are used for custom skills

```
{
  "interactionModel": {
    "languageModel": {
      "invocationName": "movie recommender",
      "intents": [{
        "name": "LikeMovieIntent",
        "slots": [{
          "name": "Movie",
          "type": "AMAZON.SearchQuery",
          }
        ],
        "samples": [
          "I like the movie {Movie}",
          "Add the movie {Movie}"
        ]
      }],
      "types": []
    }
  }
}
```

Figure 3.33.: Alexa JSON Configuration File

```
@ask.intent('LikeMovieIntent', mapping={'data': '
   Movie'})
def like_movie(data):
    speech = "my answer"
    return statement(speech).simple_card('Hi', speech
   )
```

Figure 3.34.: Implementation of an Alexa Skill

either via voice or text commands in the *Alexa Simulator* or directly on an Amazon Echo or Echo Dot device.

3.5.1. Code Structure

For connecting the API to the Alexa Skill *Flask-Ask*[40], a Flask extension is used, which is capable handling the complete communication between the Alexa Skill and the API. The main implementation for this task is done in the `recsys/alexa.py` file with one function for each Intent. With *flask-ask* it is easy to connect a function with an intent, seen in Figure 3.34.

A function is connected to an intent by annotating it with `@ask.intent` and stating the intent name. If there are any data passed from the function an additional mapping has to be defined. Then, the function expects a parameter of the same name.

Now, the function should return either `statement` or `question` together with the answer string. A question can also wait for the user's input by using the `reprompt` function, heavily used in the Alexa Skill implementation for this thesis. So the user does not have to start from the beginning and has to say "Alexa" together with the Skill's Invocation name for each statement.

Additionally, `simple_card` has to be called. This provides a visual feedback on the Alexa App of every executed Intent.

---

[40] `https://github.com/johnwheeler/flask-ask`

For this implementation the Skill's session is used to get the user id and to store the user's current page when scrolling through the result pages. Sessions are globally available via a `session` object that can be imported from *flask-ask*. One has to use the dict `session.attributes[MYKEY] = MYVAL` to store MYVAL into the MYKEY session.

When a user only says the invocation name a special launch function is called. This has to be annotated by using `@ask_launch`. For this implementation, every function as well as the launch function calls the `init` function from the `bot.py` file which initializes the RiveScript interpreter as described in Subsection 3.4.2. Therefore, any response is generated by the API the same way as it generates responses for chatbots.

The only differences for the implementation between Alexa and a chatbot are that the given answers have to be shorter, without links, less items on one page and can be formatted by using Speech Synthesis Markup Language (SSML)[82]. SSML is a markup language designed to provide meta information on text to improve the quality of it when spoken. For this implementation, `<emphasis>` to emphasize a phrase, `<break>` to add pauses and `<s>` to separate sentences, are used for example, after each item in a list. A sentence using SSML has to be surrounded by `<speak>`.

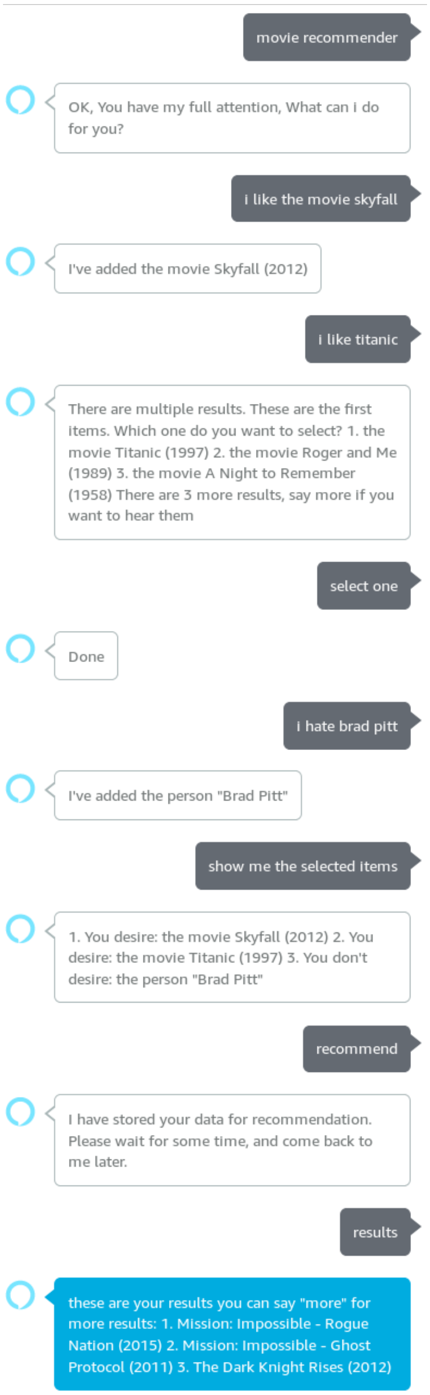
To enable that the bot returns a text prepared for Alexa, the style parameter (discussed in Subsection 3.4.2) has to be set to the value `speech` and the bot's logic takes care of the rest.

As RiveScript needs any statement from the user (e.g., "I like the movie Skyfall") but the Alexa Skill is only capable of returning the content of a Slot ("Skyfall" in this example), an artificially created statement has to be created. So for each intent the basic RiveScript command from Figure 3.30 together with a numeric value (needed for debugging purposes) is appended by the content of the Slot.

A big difference to the other interfaces is the restriction that every Skill has to reply within eight seconds, otherwise the execution of the Skill is stopped immediately. Unfortunately, there isn't a push notification API publicly available yet. However, this will be better in future as the closed developer preview has been published in November 2017 [37]. Using the

Alexa Interface the user has to request the results manually by saying `results` or `Give me the results`.

A transcript of how to use the Alexa Skill is the output of the Amazon Alexa Simulator available in Figure 3.35.



movie recommender

OK, You have my full attention, What can i do for you?

i like the movie skyfall

I've added the movie Skyfall (2012)

i like titanic

There are multiple results. These are the first items. Which one do you want to select? 1. the movie Titanic (1997) 2. the movie Roger and Me (1989) 3. the movie A Night to Remember (1958) There are 3 more results, say more if you want to hear them

select one

Done

i hate brad pitt

I've added the person "Brad Pitt"

show me the selected items

1. You desire: the movie Skyfall (2012) 2. You desire: the movie Titanic (1997) 3. You don't desire: the person "Brad Pitt"

recommend

I have stored your data for recommendation. Please wait for some time, and come back to me later.

results

these are your results you can say "more" for more results: 1. Mission: Impossible - Rogue Nation (2015) 2. Mission: Impossible - Ghost Protocol (2011) 3. The Dark Knight Rises (2012)

Figure 3.35.: Example Alexa Chat Page

4. Evaluation

This chapter describes the process of the evaluation of the three recommender interfaces implemented in Chapter 3. A user study with 16 users was performed with the aim to get insights on the users' overall satisfactions when interacting with the interfaces and measure the system's usability by using their feedback.

The *U.S. Department of Health and Human Services*[1] defines usability as: "how effectively, efficiently and satisfactorily a user can interact with a user interface." [93].

To measure usability, after letting the users try out and compare each interface, a survey was used to collect the user's impressions. This was evaluated with the *System Usability Scale (SUS)*, described in Section 4.1, the results with detailed plots illustrating the users' inputs can be found in Chapter 5, in Chapter 6 the results are examined and discussed.

4.1. System Usability Scale

The System Usability Scale (SUS) was proposed in "SUS-A quick and dirty usability scale." by John Brooke in [17] as a "quick and dirty" method to measure usability and is based on the *Likert Scale*.

---

[1] `https://hhs.gov`

4.1.1. Likert Scale

The Likert Scale is a classical measurement in the field of sociology to value the attitude of a person towards a subject and was introduced in the year 1932 [62]. It basically consists of statements where a user has to state whether he agrees or disagrees. Each statement is either clearly negative or positive.

The granularity of the steps between "agree" and "disagree" are freely definable, five are very commonly used. Having an even numbered set of choices forces the user to decide for one of the two scale's sides, when having an uneven number of choices on the scales gives the user to not decide whether they agree or not. This is a permanently discussed subject, however it Robert Armstrong shows in [13] that neither of the two choices have a statistically significant impact. Additionally John Dawes shows additionally that any of the scales with in incremental step size of 5, 7 or 10 proved to result in roughly the same mean result [29].

4.1.2. Implementation

The SUS is a Likert Scale, explicitly created for measuring usability. It uses ten statements, five positive and five negative in alternating order. These should be rated by the user within a scale from one to five where a one denotes they *strongly disagree* with the sentence and five that they *strongly agree* with the meaning of this sentence. The alternating order was introduced to ensure that users use their full attention to correctly agree or disagree [18].

The ten statements used for the System Usability Score of the user study, first introduced in [17], are as follows:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.

6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The advantage of the System Usability Score is that it is cheap and easy to implement and interpret. For users the advantage lies in being a fast, easy and uncomplicated method to give feedback. The disadvantage is that only a single score, the overall feedback for the evaluated system can be determined. The possible individual faults within a system cannot be determined directly, because the score reflects only the easiness in the system's use and therefore showing the user's overall satisfaction.

4.1.3. Results of SUS

The System Usability Score is calculated as follows:

- For all positive statements (sentence 1,3,5,7 and 9) the user's input (a number between 1 and 5) gets decremented by one.
- For the negative statements (2,4,6,8 and 10) the user's input gets subtracted from five.
- Those numbers are summed and then multiplied by 2.5.

An example implementation in Python can be seen in Figure B.1, where a function `sus_score` takes an array with the input values reflecting the user's input.

The resulting score lies in between a minimum value of 0 and a maximum value of 100. The inventor of SUS, John Brooke, explains the reasons for setting the interval between 0 and 100, which causes a slightly more complex calculation of the score, to be simply marketing reasons [18]. The resulting caveat is that it could been seen as a percentage value, as the range might indicate. However the score can only be used to compare SUS tested systems.

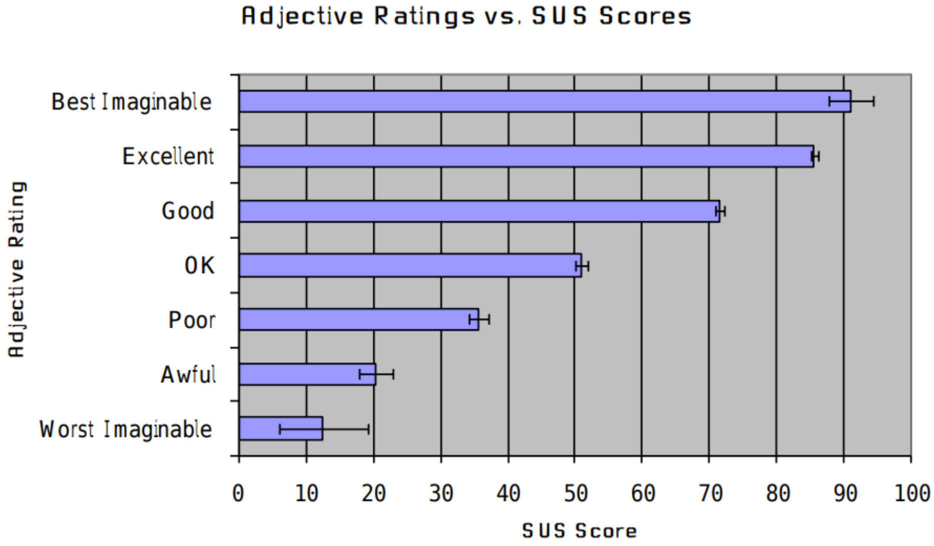


Figure 4.1.: Comparing SUS with Adjective Rating, Source: [14]

The overall number combines all the feedback gained from the user into one single score, without any additional information. The SUS can only be interpreted as a combined result so it is not possible to rely on the feedback of only one of the ten statements.

4.1.4. Interpreting the Score

While the original definition of the SUS did only state that a higher value means a higher satisfaction and therefore a higher usability of the tested system, a specific meaning of the resulting score was not added until Aaron Bangor in 2008 [14]. Another set of statements to a SUS survey was added, so-called “adjective ratings” instead of the numeric scale were used. Users were asked to choose one of the labels “Worst Imaginabel”, “Awful”, “Poor”, “OK”, “Good”, “Excellent” or “ Best Imaginable”, presented in that order. Then they compared the results to the ones of the standard SUS. The connection between SUS and those labels, that are far more achievable for humans, is shown in Figure 4.1.

The System Usability Score should never be interpreted as a percentage value, as the 50th percentile is around the scale of 68 meaning that this is the average result for the SUS [18].

When seeing up a SUS evaluated user study, it is important to not choose overly difficult tasks as that might influence the score negatively. On the other hand it has been shown that SUS gives reliable results even when the user had only a very short period of time to familiarize themselves with the system under test. The score of users who used the evaluated system for only five seconds has been shown to only differ by three points in the overall average from those who did not get a time restriction [2]. This means that the first impression of a system does not differ that much from the experience a user reports later [18].

Thomas Tullis states that the SUS gives good results even when evaluating with a small set of participants [91]. They conclude that the scores for eight participants are already quite good with an accuracy of 75%, but for twelve they are nearly indistinguishable from the survey of the full dataset with an accuracy of 90-100%.

In summary, the System Usability Scale is a well-researched tool for evaluating the usability of user interfaces. Even the constraints, such as a short system testing time and having a small number of participants, does not necessarily influence the results negatively. Therefore, the System Usability Scale is an ideal reliable measurement for the user study of the different movie recommender interfaces to reflect the user's satisfaction. The results for this study are shown in Chapter 5.

4.2. User Study

The user study took place between 12th April 2018 and 21st April 2018 with sixteen participants, 22 invitations were sent.

The study consisted of a separate task for each tested movie recommender interface and a concluding questionnaire. Each user had to test at least the webform and the chatbot interface. The third, the virtual assistance system implemented by an Alexa Skill, was optional as the participants were not expected to own an Alexa enabled device like the Echo Dot—which is necessary for testing the Alexa Skill. They were given the chance to test it with a provided device. Ten out of the sixteen participants agreed on testing and evaluating the third interface as well.

The users could either evaluate the interfaces remotely in their own home with their preferred choice of devices or take the user study in a specifically set up space Subsection 4.2.1.

All participants got an introduction, either via a document or locally, into the general topic and the purpose of the user study, reminding them that the usability of the interfaces and not the individual user and their knowledge or performance was being evaluated. Additionally they were asked not the evaluate the results of the recommender itself, as this is not the focus of this study.

The instructions for those who tested the Alexa Skill can be found in Figure B.2, while the version for the minimal test setup is Figure B.3, each containing an adjusted survey and description.

4.2.1. Setup

A distinct lab setup, especially for participants of the Alexa interface was provided. They could choose between an *Archlinux*[2] desktop computer or a MacBook Pro 2015. Google's *Chrome*[3], *Chromium*[4] or *Firefox*[5] as web browser were available as well as a test account for the Facebook Messenger Chat and for Telegram. The Alexa interface was tested by using an Amazon Echo Dot 2nd Generation provided by the Institute of Interactive Systems and and Data Science.

Participants testing only the webform and the chatbot interface could use their preferred device or browser. However, due to the restrictions on the

---

[2] `https://archlinux.org`

[3] `https://www.google.at/chrome/`

[4] `https://chromium`

[5] `https://firefox.com`

Facebook Messenger[6] login data for a Facebook test account were provided when a user wanted to use Facebook Messenger.

4.2.2. Tasks

The underlying goal of the user study was to let the participants familiarize with each interface and capture their feedback on the usability.

The given tasks were meant as a possibility to get an outline of all provided features so that the participants got a guideline to follow through. The list of the tasks can be found in the invitation (Figure B.2 for Alexa testers and Figure B.3 for webform/chatbot only testers). The tasks provided were the same for each interface for better comparison of the evaluation results.

4.2.3. Survey

Two different surveys were prepared, one for the minimal test set of webform and chatbot and one for those testing the Alexa Skill. The survey itself was done with *Google Forms*[7].

First the participants had to fill in a set of basic data about themselves such as, gender, age, highest education, and the self estimation of their computer knowledge ranging from "Never used a computer" to "Expert" level. Additionally, they were asked about previous experiences with the technologies used for the different recommender interfaces. They could state if they ever used a chatbot or a speech recognition software (for Alexa testers only).

After that, for each tested interface they had to state the used device followed by the statements for the System Usability Score and completed by a free input field letting the participants input any additional remarks.

---

[6] a user has to have an active developer account

[7] `https://forms.google.com`

A screenshot of the first page of the survey is shown in Figure B.4, while Figure B.5 and Figure B.6 display the survey page for the webform with the questions for the System Usability Score.

This chapter first discussed the System Usability Score and concluded that it is a reliable tool for measuring the overall satisfaction of a user interface. Then the executed user study to get the user's feedback was described. The following chapter shows and examines the results of the study.

5. Results

In the previous chapter the theoretical background and implementation of three different interfaces for a movie recommender and the basic principles of a user study were discussed. Insights into the users' impressions were gained with a survey that followed their evaluation of the available interfaces. These results are described in this chapter.

5.1. User Study Participants

22 persons were invited to participate in this user study. 16 of them agreed to participate.

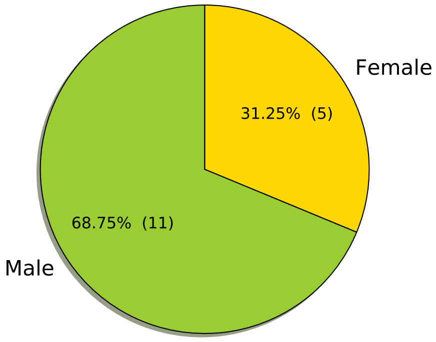


Figure 5.1.: Participants by gender

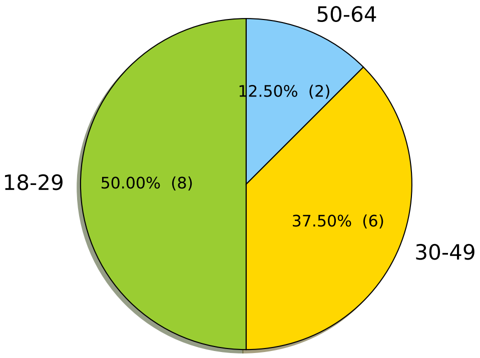


Figure 5.2.: Participants by age

Figure 5.1 shows that the ratio between female and male participation is about one to three and Figure 5.2 shows that half of the participants were between the age of 18 to 29, only two participants between the age of 50 and 64, and the remaining were between 30 and 49 years old.

Figure 5.3 reflects the participants' highest level of education. More than 43% of them had a degree of a university while a bit more than 37% had finished High School. Only three participants (18.75%) had finished other types of education like an apprenticeship. Users were asked to self assess their level of confidence with technology. Nearly two third described themselves as competent and one fourth considered themselves to be experts. Two participants thought of themselves to be at a beginner level.

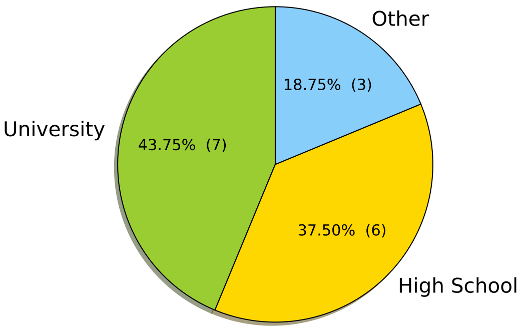


Figure 5.3.: Participants by education

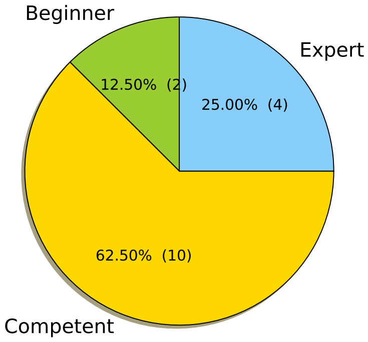


Figure 5.4.: Participants by knowledge

In this study four users (25%) had already used a chatbot (see Figure 5.5), but only half of that amount, two users, had used Alexa, Siri or other voice command based systems before (see Figure 5.6).

The next section analyses the questions from the survey resulting in the System Usability Score in more detail.

5.2. Comparison by Interface

The original goal of this thesis was to find out more about the usability of recommender interfaces. In Figure 5.7 the final System Usability Score,

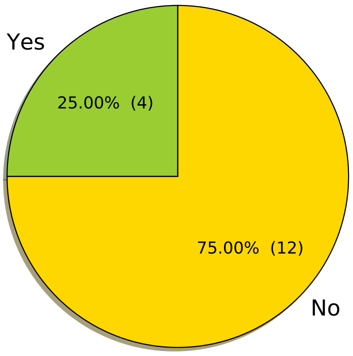


Figure 5.5.: Participants with prior chatbot experience

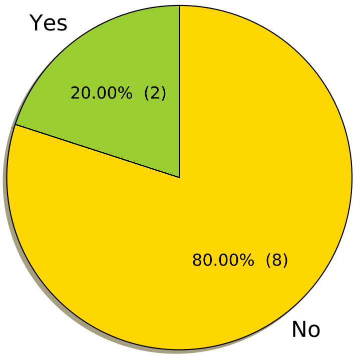


Figure 5.6.: Participants with prior speech recognition experience

rounded to the next full number, for each interface is displayed. The webform reaches an average score of 64 out of 100, which is almost near the average rating for the scale, as discussed in Section 4.1,. The webform's highest score given from one user is 92 and the lowest 32. Therefore the Adjective Rating of the webform would be considered as "Good", but also already seen as "Best Imaginable" by some users.

The chatbot interface got a score ranging from 5 to 82, therefore some users considered it to be "Excellent" while others could not use it at all and thought of its status as "Worst Imaginable". The average score for the chatbot interface is 44 which is clearly lower than the score for the webform. This would give the second interface the label "OK" on the Adjective Rating scale.

The third evaluated interface, voice recognition with Alexa, has a slightly higher average score than the chatbot interface, with 53 which is still lower than the webform. The minimum score given is 22 ("Awful") which is still higher than the minimum of the chatbot. Alexa's maximum is 87, therefore labelling the Alexa interface even as "Excellent" in the eyes of some users.

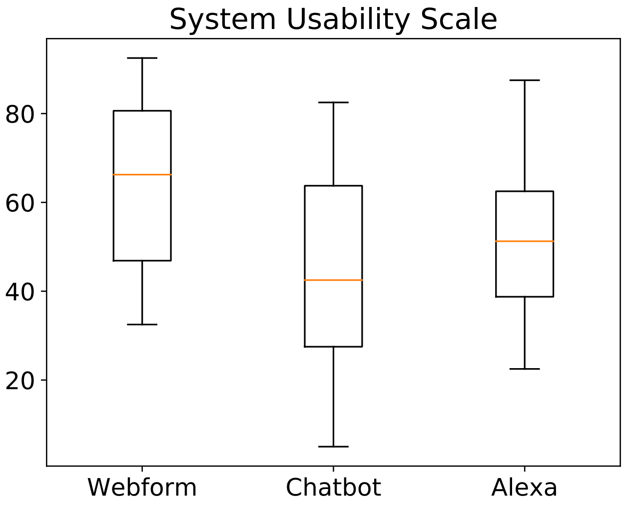


Figure 5.7.: Overall comparison of all interfaces

5.2.1. Detailed System Usability Scores

An immediately rising question when looking at the range of results is, if there are any significant differences in the usability one could attribute to differences of gender, age, prior knowledge or understanding of computers.

In Figure 5.8 the System Usability Score is split by gender. While for the webform both groups seem to agree, both chatbot and Alexa were quite different, males preferred Alexa, and females preferred the chatbot.

The System Usability Score of the interfaces separated by age can be seen in Figure 5.9. It is noticeable that the older participants (older than 50 years) were displeased with the webform and the Alexa Skill, but had an above-average rating for the chatbot interface.

In Figure 5.10 the System Usability Score is split by the user's education. Here, it can be observed that participants with an university degree really liked the webform.

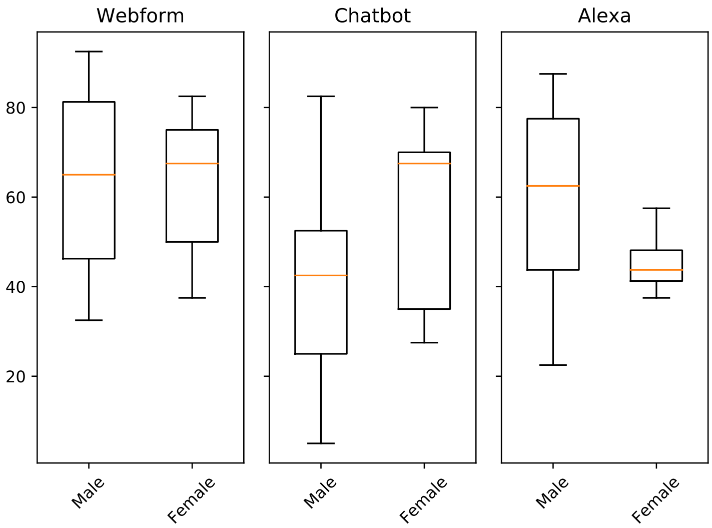


Figure 5.8.: SUS by Gender

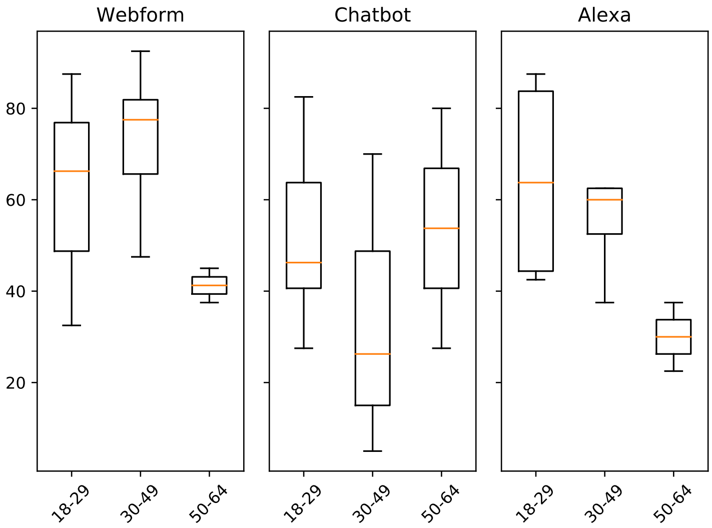


Figure 5.9.: SUS by Age

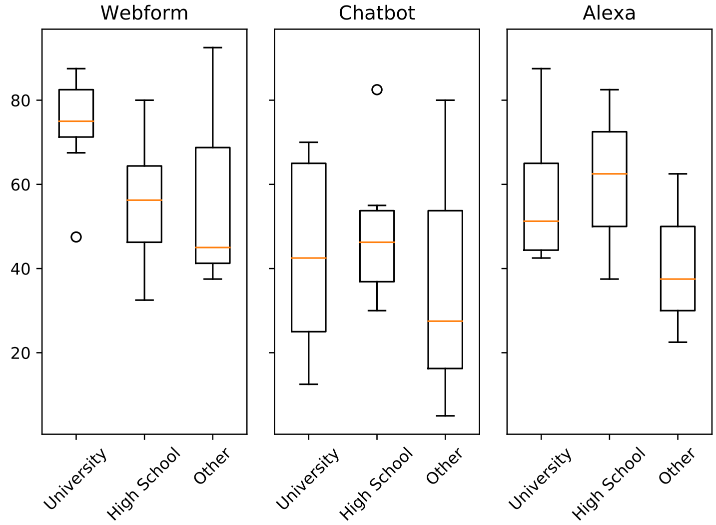


Figure 5.10.: SUS by Education

When separating the data by the participant's computer knowledge level it is visible that experts liked the webform and beginner did not like working with the Echo Dot.

The complete set of results, with all data, can be seen in Figure 5.13, with the minimum, average, and maximum System Usability Score for each tested interface.

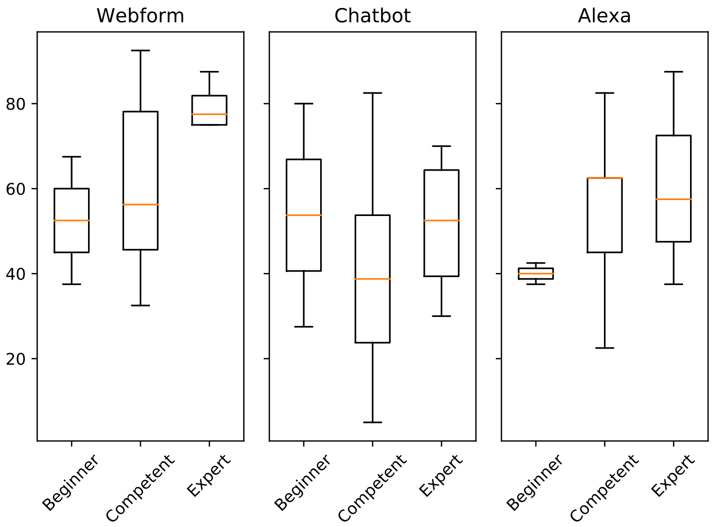


Figure 5.11.: SUS by Computer Knowledge

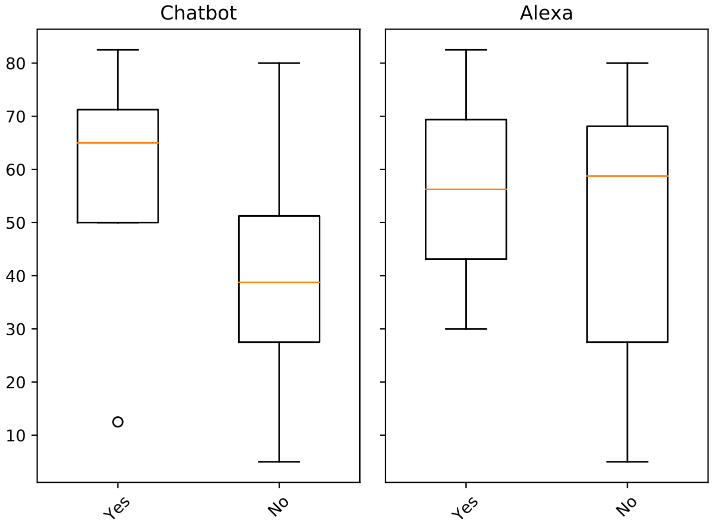


Figure 5.12.: SUS by previous usage

	Webform	**Chatbot**	**Alexa**
		TOTAL	
	(92) 64 (32)	(82) 44 (5)	(88) 53 (23)
		By Gender	
male	(93) 65 (33)	(83) 39 (5)	(88) 59 (23)
female	(83) 63 (38)	(80) 56 (27)	(58) 46 (38)
		By Age	
18-29	(88) 63 (33)	(83) 51 (28)	(88) 65 (23)
30-49	(93) 73 (48)	(70) 32 (5)	(63) 55 (38)
50-64	(45) 41 (38)	(80) 54 (28)	(38) 30 (23)
		By Education	
University	(88) 74 (48)	(70) 44 (13)	(88) 58 (43)
High School	(80) 56 (33)	(83) 50 (30)	(83) 61 (38)
Other	(93) 59 (38)	(80) 38 (5)	(63) 41 (23)
		By Knowledge	
Beginner	(68) 53 (38)	(80) 54 (28)	(43) 40 (38)
Competent	(93) 61 (33)	(83) 40 (5)	(83) 55 (23)
Expert	(88) 79 (75)	(70) 51 (30)	(88) 61 (38)
		By Previous Chatbot Usage	
yes	—	(83) 56 (13)	—
no	—	(80) 41 (5)	—
		By Previous Alexa Usage	
yes	—	—	(83) 56 (30)
no	—	—	(80) 50 (5)

Figure 5.13.: The detailed SUS results in the format: (minimum) average (maximum), including the outliers

6. Discussion

The participants of the user study preferred the web interface over voice recognition over the chatbot interface as shown in Chapter 5. The following sections are going to investigate possible reasons for that, followed by a detailed discussion of the results by interfaces with the user's feedback (Section 6.2 for the webform, Section 6.3 for the chatbot and Section 6.4 for the Alexa Skill). Possible changes for each interface are discussed to enhance the user experience by improving the usability.

6.1. Participants

For more representative results Figure 5.2 shows that younger participants should be included and more senior participants—older than 50 years—should be included too, to reflect the average population more realistically. Likewise, as there are more than 50% females in Austria[1] [72], they are underrepresented in this user study as Figure 5.1 shows.

The educational statistics of Austria shows that 18.2% were registered in 2018 having a tertiary education. Therefore, with 43% of the participants having an academic degree, they are over represented in this study at the expense of users who are not that well educated. The same holds true for their denoted knowledge of computers as only two are seeing themselves to be a beginner.

The numbers of the participants who already used a chatbot or a speech recognition system prior to the user study, shown in Figure 5.5 and Figure 5.6 was quite expectable as chats have been around for a longer time than voice

---

[1] where this study took place

recognition systems, which were established in the customer's households in 2010 when Apple released Siri. As of today, eight years later, they are still not that commonly used.

6.2. Webform

The System Usability Score of the webform was quite expected with three exceptions: (i) older people were not satisfied, (ii) users with a lower education tend not to be that satisfied with the webform than people with a higher degree and (iii) the higher the user's computer knowledge the higher the average SUS.

Evaluating the survey's free feedback field for the webform shows that the overall rating for the web interface could have been better when changing two things: adding a button that, when pushed, deletes all the previously entered data and restarts the web application. Second, the "plus"/"minus" buttons stating the user likes or dislikes for an already selected item were not that obvious. One idea could be to modify the "Select" button from the search column to explicitly give the opportunity to "like" or "dislike" an item when selecting it.

Another good feedback given was that the search itself is rather weak, as it returns only items with a title that matches the searched word. Here it was expected that the search returns movies fitting the input in a broader way by evaluating meta data as well. For example a search for "Julia Roberts" should not only include her as an actor, but also all of the movies she acted in. Currently only genres, actors, and movies containing the word "Julia Roberts" would be returned.

The buttons to support selecting the genre or year span in the selected items column of the webform were confusing for some participants as well as they were mentioned to be "misleading" or "unintuitive". One user stated that they were not sure why those buttons are not in the search column.

Some users explicitly mentioned that refining the searched items with the buttons from the search column was a great idea and that using the webform was "easy" or "OK".

It is not surprising that the webform got a rather good rating, as many people are used to this type of web service. In the end, the SUS reflected the testers' feedback as they were rather satisfied with using this interface. Minor bugs have to be fixed so that the users are really satisfied, especially users who are older or less proficient when using a computer.

6.3. Chatbot

Overall, using the chatbot interface was considered to be more displeasing. However, looking at the complete set of data it seems that users either liked or hated this system, because the scores range from 5 to 82 which fills nearly every possible value in the scale.

Female participants were more satisfied with the chatbot interface than males, but not significantly. The differences between the various age groups, education and knowledge level are not remarkable as they reflect the overall trend.

However, looking at the System Usability Score for users who did already use a chatbot versus those who didn't, gives an interesting insight: they were satisfied using the bot, as their score is higher than the SUS's average while those who never used such a system showed the same score as the overall statistics.

There might be two reasons for this: for many users it seemed rather complicated that the webform's visual separation in three columns (searched, selected, recommended) are not directly visible. Further, most of the problems the users had were caused by the limited communication and the structure of the commands.

Communication was mentioned to be "complicated", one user explicitly remarked. Another user added that having only a limited range of words or interaction possibilities of the chatbot is both good and bad at the same time. One simply says "Couldn't really work with it". For example, it was often not understood by the participants that the commands "upvote" or "select" need a valid numerical index from an item of a previous printed list. This should have been clearer communicated by the bot, as any input that not matched any of the commands were used to search for any items (movie, genre, actor) with similar name. This was done as a user were expected just to write, for example, a movie title, but was a drawback in this scenario.

Users who were used to chatbots might therefore have been used to a limited language and specific sentence structure in order to achieve valid input. A possible improvement idea for that is to either completely restrict the language to some predefined commands, or to implement a more advanced and intelligent conversating chatbot and combine the output of multiple possible outcomes. For the example above the bot could say that upvoting the specific input is not possible but some items were found matching the user's input.

In general, users wished that the chatbot gives more information and is more transparent with the internal state of, for example, the selected items. On the other hand the feedback the bot gave was considered to be too long for some users. Additionally, the good help functionality was emphasized by some users. It can be seen that communication with a chatbot has to be carefully designed in terms of what information is given at what specific point of time during a conversation.

To summarize, the chatbot interface was either liked or hated by the users which can be seen from the System Usability Score and is confirmed by the users' remarks. A lot of work has to be done to make the chatbot interface user friendly, mainly improving the communication itself, for example, by replacing pattern matching by more advanced techniques.

6.4. Alexa

Using the Alexa interface led to more satisfaction than the chatbot interface. At first this seems to be counter intuitive as the Alexa interface's implementation uses the chatbot's logic in the background. However, there are certain factors that might have caused this shift. First, Alexa was tested after the chatbot, so the user already knew how to formulate certain commands. Second, the act of speaking to a computer was new to most of the participants and the curiosity for new gadgets might cause the users to overlook some problems with this interface.

The main problem was, due to the nature of the device, that no complete reference for the implemented commands is available. So it was added to the introduction for the participants, additionally, the help command could have been used. The participants often tended to forget the mentioned commands. Another relevant problem with Alexa is however the sensibility of the voice recognition. It has been observed that even the invocation name “Movie Recommender” has often not been detected correctly by the system, even with a very clear pronunciation. On the other hand some users could start the movie recommender on the Echo Dot without any problems even with mispronunciations. A flaw that occurs when Alexa does not understand the user correctly is that the system seems to guess what was said. This leads sometimes to awkward and wrong results.

Another problem is that Alexa enforces a fast reaction. The Alexa Skill will stop when there was no interaction for more than eight seconds. This led to a lot of frustration as than the user had to start again with the trigger word for Alexa and the invocation name “Movie Recommender”. Generally users noted that there was no thinking pause allowed, and they felt pressured or overstrained to give a command in that short time.

It can be assumed that the chatbot interface was not that user friendly as a free conversation is not possible due to pattern matching, but this was what most of the users expected.

The Alexa Skill clearly is rule based and therefore the commands are clearly defined contrary to the chatbot. This might have been one of the reasons of the higher System Usability Score for the Alexa Skill.

To improve the satisfaction the main work here has to be done by Amazon as their voice recognition often gives unexpected results. Amazon should also provide a push notification system, so that the users do not have to ask Alexa for the results themselves. The other improvements that should be implemented are mainly the same as with the other tasks: improving the search by allowing requests for metadata and improving the communication.

7. Conclusions and Future Work

This thesis evaluates three different recommender interfaces: webform, chatbot and virtual assistant. A user study is described that was executed with 16 persons with the overall result, that users preferred the webform over voice recognition and over the chatbot.

This shows that much work has to be done to establish chatbots for a broader audience. However, those users who had used chatbots previously had a higher acceptance for that interface as those who don't. This shows that although chatbots are available all over the Internet, users are still rather unfamiliar with them.

For virtual assistance systems users showed that the main problem has been in the system's understanding of the spoken words. Although the overall quality has been quite good with Alexa, sometimes the system could not understand a clear and good pronunciation while it could sometimes understand mumbled words with a heavy dialect. When eliminating those problems the acceptance of Alexa would clearly be much higher.

The implemented chatbot has one drawback, as a natural conversation is not possible due to the nature of pattern matching. So for communication users have to use a more commanding style with a fixed task in mind instead of talking freely. This approach turned out to be too limited on many occasions while for the virtual assistance system Alexa, this approach comes naturally. Users obvious expect to give commands to such a system and therefore the overall satisfaction was quite higher.

This effect of being used to a certain way of communication is it why the overall satisfaction of the webform is so much higher than with the other interfaces, as webforms are all over the Internet for many years while the other technologies, especially the virtual assistance systems are not that widespread yet.

Future work could include a more rich-text chatbot behaviour, so that the chatbot does not only communicate by text, but also can use advanced widgets that might significantly improve the quality.

Another more general idea is to extend the search for movies, actors etc in a more contextual sense, so not to include only direct results of the user's input, but also items that are related to the input. So a search for "Brad Pitt" should not only deliver the actor "Brad Pitt" or any movie where the title includes his name, but also all the movies where he acted in.

The main research however will be to include machine learning or other more advanced techniques to chatbots and voice based assistance systems. The results of the user study show, that chatbots that are solely based on pattern matching are not good enough to really satisfy users, therefore research has to be done to enable a more open-domain approach for communication. Here machine learning techniques including neural networks are an obvious but also a very promising idea that should be investigated.

Appendix

Appendix A.

Appendix: Implementation

```
genre:
+-----------+------------------+------+-----+---------+----------------+
| Field     | Type             | Null | Key | Default | Extra          |
+-----------+------------------+------+-----+---------+----------------+
| id        | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| genrename | varchar(45)      | NO   | UNI | NULL    |                |
+-----------+------------------+------+-----+---------+----------------+

movie__matched:
+---------------------+------------------+------+-----+---------+-------+
| Field               | Type             | Null | Key | Default | Extra |
+---------------------+------------------+------+-----+---------+-------+
| id                  | int(10) unsigned | NO   | PRI | NULL    |       |
| ttid                | varchar(10)      | NO   | UNI | NULL    |       |
| title               | varchar(191)     | YES  | UNI | NULL    |       |
| plot                | text             | YES  |     | NULL    |       |
| rating_votes        | int(11)          | YES  |     | NULL    |       |
| rating_rank         | decimal(4,2)     | YES  |     | NULL    |       |
| rating_distribution | varchar(11)      | YES  |     | NULL    |       |
| year                | int(11)          | YES  |     | NULL    |       |
| running_time        | int(11)          | YES  |     | NULL    |       |
+---------------------+------------------+------+-----+---------+-------+

movie__matched_alternative_title
+-------------------+------------------+------+-----+---------+-------+
| Field             | Type             | Null | Key | Default | Extra |
+-------------------+------------------+------+-----+---------+-------+
| movie_id          | int(10) unsigned | NO   | MUL | NULL    |       |
| alternative_title | varchar(600)     | NO   | UNI | NULL    |       |
| info              | text             | YES  |     | NULL    |       |
+-------------------+------------------+------+-----+---------+-------+

movie_person
+-----------+------------------+------+-----+---------+-------+
| Field     | Type             | Null | Key | Default | Extra |
+-----------+------------------+------+-----+---------+-------+
| movie_id  | int(11) unsigned | NO   | PRI | NULL    |       |
| role_id   | int(11) unsigned | NO   | PRI | NULL    |       |
| person_id | int(11) unsigned | NO   | PRI | NULL    |       |
+-----------+------------------+------+-----+---------+-------+

person
+-------------------+------------------+------+-----+---------+----------------+
| Field             | Type             | Null | Key | Default | Extra          |
+-------------------+------------------+------+-----+---------+----------------+
| id                | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| first_name        | varchar(45)      | YES  | MUL | NULL    |                |
| last_name         | varchar(45)      | YES  |     | NULL    |                |
| nr                | varchar(45)      | YES  |     | NULL    |                |
| gender            | varchar(2)       | YES  |     | NULL    |                |
| biography         | text             | YES  |     | NULL    |                |
| dob               | date             | YES  |     | NULL    |                |
| dod               | date             | YES  |     | NULL    |                |
| location_birth_id | int(10) unsigned | YES  | MUL | NULL    |                |
| location_death_id | int(10) unsigned | YES  | MUL | NULL    |                |
+-------------------+------------------+------+-----+---------+----------------+
```

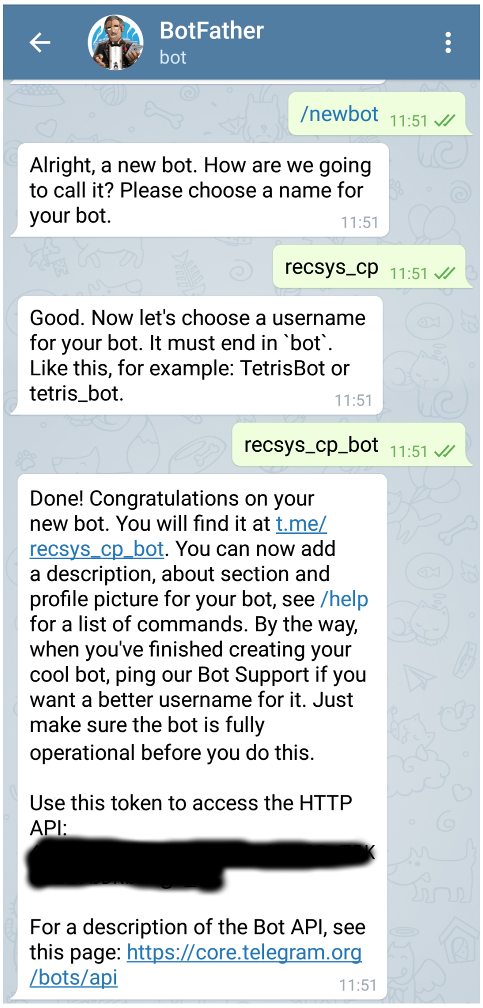
Figure A.1.: Database structure of the IMDb database

```
alternative_genre:
+------------------+-------------+------+-----+---------+----------------+
| Field            | Type        | Null | Key | Default | Extra          |
+------------------+-------------+------+-----+---------+----------------+
| id               | int(11)     | NO   | PRI | NULL    | auto_increment |
| alternative_name | varchar(45) | YES  |     | NULL    |                |
| orig_id          | int(11)     | YES  |     | NULL    |                |
| orig_name        | varchar(45) | YES  |     | NULL    |                |
+------------------+-------------+------+-----+---------+----------------+

bot:
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| user      | varchar(255) | NO   | PRI |         |       |
| search    | mediumtext   | YES  |     | NULL    |       |
| selected  | mediumtext   | YES  |     | NULL    |       |
| page      | int(11)      | YES  |     | NULL    |       |
| timestamp | datetime     | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+

bot_log:
+-----------+------------------+------+-----+---------+----------------+
| Field     | Type             | Null | Key | Default | Extra          |
+-----------+------------------+------+-----+---------+----------------+
| id        | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| user      | varchar(255)     | YES  |     | NULL    |                |
| provider  | varchar(255)     | YES  |     | NULL    |                |
| request   | text             | YES  |     | NULL    |                |
| reply     | text             | YES  |     | NULL    |                |
| timestamp | datetime         | YES  |     | NULL    |                |
+-----------+------------------+------+-----+---------+----------------+

log:
+-----------+------------------+------+-----+---------+----------------+
| Field     | Type             | Null | Key | Default | Extra          |
+-----------+------------------+------+-----+---------+----------------+
| id        | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| session   | char(36)         | YES  |     | NULL    |                |
| client    | varchar(255)     | YES  |     | NULL    |                |
| data      | longtext         | YES  |     | NULL    |                |
| timestamp | datetime         | YES  |     | NULL    |                |
+-----------+------------------+------+-----+---------+----------------+

person:
+------------+------------------+------+-----+---------+-------+
| Field      | Type             | Null | Key | Default | Extra |
+------------+------------------+------+-----+---------+-------+
| id         | int(10) unsigned | NO   | PRI | NULL    |       |
| rank       | int(11)          | YES  |     | NULL    |       |
| first_name | varchar(45)      | YES  |     | NULL    |       |
| last_name  | varchar(45)      | YES  |     | NULL    |       |
| name       | varchar(255)     | YES  |     | NULL    |       |
+------------+------------------+------+-----+---------+-------+

recommendations:
+----------------------------+------------------+------+-----+---------+----------------+
| Field                      | Type             | Null | Key | Default | Extra          |
+----------------------------+------------------+------+-----+---------+----------------+
| id                         | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| client                     | varchar(255)     | YES  |     | NULL    |                |
| session                    | varchar(255)     | YES  |     | NULL    |                |
| uuid                       | varchar(255)     | YES  |     | NULL    |                |
| items                      | mediumtext       | YES  |     | NULL    |                |
| items_raw                  | mediumtext       | YES  |     | NULL    |                |
| recommendation             | mediumtext       | YES  |     | NULL    |                |
| created                    | datetime         | YES  |     | NULL    |                |
| rec_log_table_id           | int(10)          | YES  |     | NULL    |                |
| timestamp_start            | timestamp        | YES  |     | NULL    |                |
| timestamp_start_processing | timestamp        | YES  |     | NULL    |                |
| timestamp_end              | timestamp        | YES  |     | NULL    |                |
| bot_user                   | varchar(255)     | YES  |     | NULL    |                |
| sent_to_bot                | tinyint(1)       | YES  |     | NULL    |                |
| bot_provider               | varchar(100)     | YES  |     | NULL    |                |
+----------------------------+------------------+------+-----+---------+----------------+
```

Figure A.2.: Database structure of the cpaier database, Part 1

```
vote_log:
+----------------+--------------+------+-----+---------+-------+
| Field          | Type         | Null | Key | Default | Extra |
+----------------+--------------+------+-----+---------+-------+
| ip             | varchar(15)  | NO   | PRI |         |       |
| movie_id       | int(11)      | NO   | PRI | 0       |       |
| session        | char(36)     | YES  |     | NULL    |       |
| client         | varchar(255) | YES  |     | NULL    |       |
| direction      | varchar(4)   | YES  |     | NULL    |       |
| recsys_session | varchar(255) | YES  |     | NULL    |       |
| timestamp      | datetime     | YES  |     | NULL    |       |
+----------------+--------------+------+-----+---------+-------+

vote:
+----------+------------------+------+-----+---------+-------+
| Field    | Type             | Null | Key | Default | Extra |
+----------+------------------+------+-----+---------+-------+
| movie_id | int(10) unsigned | NO   | PRI | NULL    |       |
| voting   | int(11)          | YES  |     | NULL    |       |
+----------+------------------+------+-----+---------+-------+
```

Figure A.3.: Database structure of the cpaier database, Part 2



/newbot 11:51

Alright, a new bot. How are we going to call it? Please choose a name for your bot. 11:51

recsys_cp 11:51

Good. Now let's choose a username for your bot. It must end in `bot`. Like this, for example: TetrisBot or tetris_bot. 11:51

recsys_cp_bot 11:51

Done! Congratulations on your new bot. You will find it at t.me/recsys_cp_bot. You can now add a description, about section and profile picture for your bot, see /help for a list of commands. By the way, when you've finished creating your cool bot, ping our Bot Support if you want a better username for it. Just make sure the bot is fully operational before you do this.

Use this token to access the HTTP API:
[redacted]

For a description of the Bot API, see this page: https://core.telegram.org/bots/api 11:51

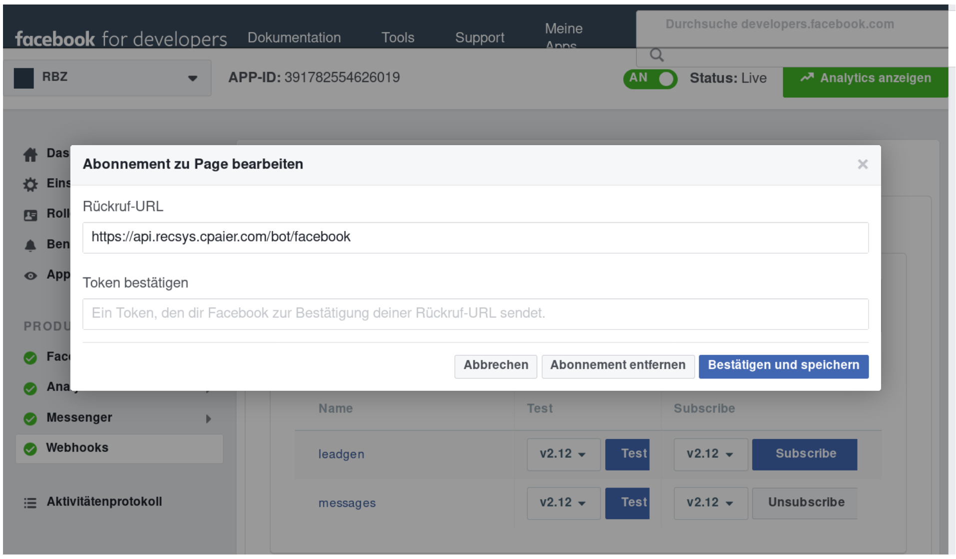
Figure A.4.: Initiate telegram chatbot

```
{'message': {
  'date': 1523639599,
  'text': 'hi',
  'from': {
    'first_name': 'Christian',
    'is_bot': False,
    'id': 155534818,
    'language_code': 'de-AT'
  },
  'message_id': 2039,
  'chat': {
    'first_name': 'Christian',
    'type': 'private',
    'id': 155534818
  }
}, 'update_id': 716293453}
```

Figure A.5.: Incoming Telegram Message

```
data = {
  "chat_id": chatid,
  "text": answer_message
}
url = "https://api.telegram.org/bot"+TOKEN+"/
   sendMessage"
requests.post(url, data=data)
```

Figure A.6.: Sending messages for Telegram with Python



Abonnement zu Page bearbeiten

Rückruf-URL

https://api.recsys.cpaier.com/bot/facebook

Token bestätigen

Ein Token, den dir Facebook zur Bestätigung deiner Rückruf-URL sendet.

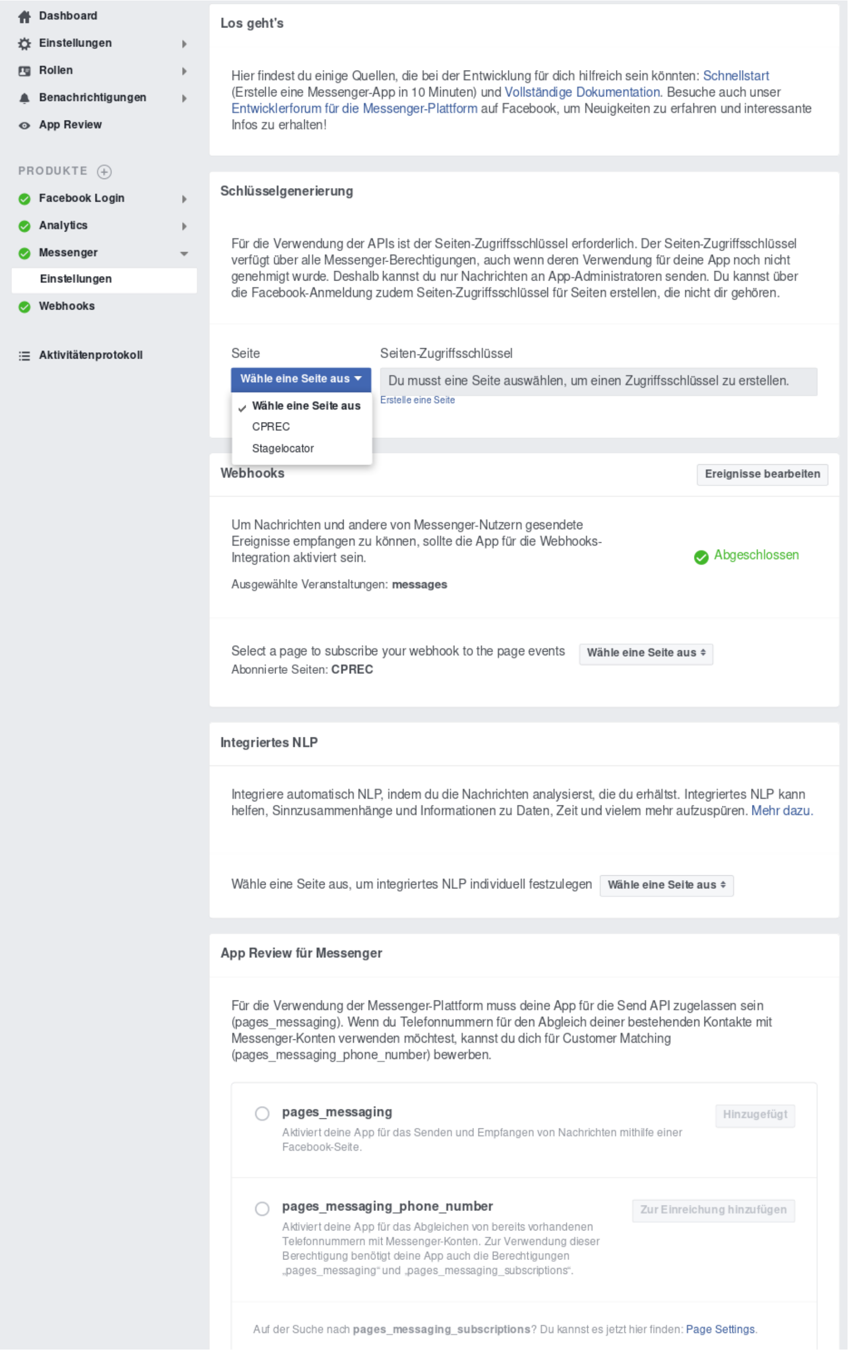
Figure A.7.: Facebook: setting a webhook

```
{"entry": [{
    "messaging": [
      {
        "timestamp": 1523827339183,
          "message": {
            "text": "hi",
            "mid": "mid.$cAADEB–YKsQNo_HU3r1iyy17wuvk8",
            "seq": 2054
          },
          "recipient": {
            "id": "1747717198342l5"
          },
          "sender": {
            "id": "2044173955599901"
          }
        }
      ],
      "id": "1747717198342l5",
      "time": 1523827339457
  }],
  "object": "page"
}
```

Figure A.8.: Incoming message Facebook Messenger

```
def facebook_answer(key, sender_id, answer):
   params = { "access_token": key }
   headers = { "Content-Type": "application/json"}
   data = json.dumps({
       "recipient": {
           "id": sender_id
       },
       "message": {
           "text": answer
       }
   })

   url="https://graph.facebook.com/v2.6/me/messages"
   requests.post(url, params=params, headers=headers
  , data=data)
```

Figure A.9.: Replying to Facebook Messenger



Los geht's

Hier findest du einige Quellen, die bei der Entwicklung für dich hilfreich sein könnten: Schnellstart (Erstelle eine Messenger-App in 10 Minuten) und Vollständige Dokumentation. Besuche auch unser Entwicklerforum für die Messenger-Plattform auf Facebook, um Neuigkeiten zu erfahren und interessante Infos zu erhalten!

Schlüsselgenerierung

Für die Verwendung der APIs ist der Seiten-Zugriffsschlüssel erforderlich. Der Seiten-Zugriffsschlüssel verfügt über alle Messenger-Berechtigungen, auch wenn deren Verwendung für deine App noch nicht genehmigt wurde. Deshalb kannst du nur Nachrichten an App-Administratoren senden. Du kannst über die Facebook-Anmeldung zudem Seiten-Zugriffsschlüssel für Seiten erstellen, die nicht dir gehören.

Seite: Wähle eine Seite aus
Seiten-Zugriffsschlüssel: Du musst eine Seite auswählen, um einen Zugriffsschlüssel zu erstellen.

Webhooks

Um Nachrichten und andere von Messenger-Nutzern gesendete Ereignisse empfangen zu können, sollte die App für die Webhooks-Integration aktiviert sein.

✓ Abgeschlossen

Ausgewählte Veranstaltungen: **messages**

Select a page to subscribe your webhook to the page events: Wähle eine Seite aus
Abonnierte Seiten: **CPREC**

Integriertes NLP

Integriere automatisch NLP, indem du die Nachrichten analysierst, die du erhältst. Integriertes NLP kann helfen, Sinnzusammenhänge und Informationen zu Daten, Zeit und vielem mehr aufzuspüren. Mehr dazu.

Wähle eine Seite aus, um integriertes NLP individuell festzulegen: Wähle eine Seite aus

App Review für Messenger

Für die Verwendung der Messenger-Plattform muss deine App für die Send API zugelassen sein (pages_messaging). Wenn du Telefonnummern für den Abgleich deiner bestehenden Kontakte mit Messenger-Konten verwenden möchtest, kannst du dich für Customer Matching (pages_messaging_phone_number) bewerben.

○ **pages_messaging**
Aktiviert deine App für das Senden und Empfangen von Nachrichten mithilfe einer Facebook-Seite.

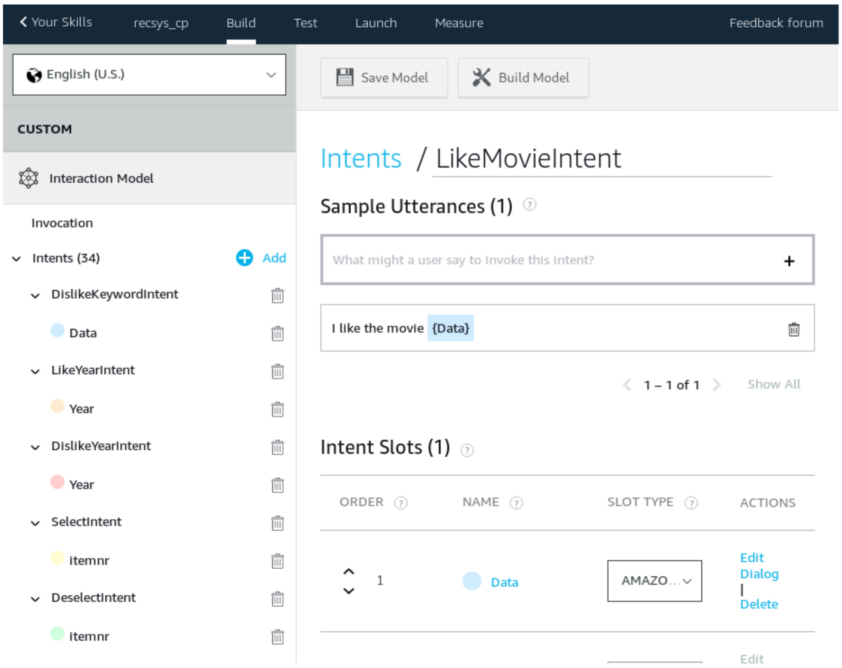
○ **pages_messaging_phone_number**
Aktiviert deine App für das Abgleichen von bereits vorhandenen Telefonnummern mit Messenger-Konten. Zur Verwendung dieser Berechtigung benötigt deine App auch die Berechtigungen „pages_messaging" und „pages_messaging_subscriptions".

Auf der Suche nach **pages_messaging_subscriptions**? Du kannst es jetzt hier finden: Page Settings.

Figure A.10.: Facebook: messenger setting

```
├── recsys
│   ├── ...
│   ├── bot.py
│   ├── brain
│   │   ├── actor.rive
│   │   ├── base.rive
│   │   ├── begin.rive
│   │   ├── functions.rive
│   │   ├── genre.rive
│   │   ├── help.rive
│   │   ├── item.rive
│   │   ├── keyword.rive
│   │   ├── movie.rive
│   │   ├── select.rive
│   │   ├── vote.rive
│   │   └── year.rive
│   └── ...
└── ...
```

Figure A.11.: Files and directories for the chatbot



I like the movie {Data}

1 Data AMAZO...

Figure A.12.: Alexa Skill Console - Modify an Intent

```
[uwsgi]
plugins        = python2
chdir          = /srv/www/api.recsys.cpaier.com/
socket         = /var/run/uwsgi/recsysapi.sock
master         = true
sharedarea     = 4
processes      = 4
vhost          = true
chmod-socket   = 777
chown-socket   = cpaier
uid            = cpaier
gid            = cpaier
logto          = /tmp/uwsgi_recsys.log
venv           = /srv/www/api.recsys.cpaier.com/venv
callable       = app
wsgi-file      = wsgi.py
```

Figure A.13.: Basic uwsgi config

```
upstream recsysapi {
   server unix:/var/run/uwsgi/recsysapi.sock;
}

server {
  listen        443 ssl;
  server_name   api.recsys.cpaier.com;
  root          /srv/www/api.recsys.cpaier.com/recsys;
  access_log    /var/log/nginx/api.recsys.cpaier.com.access.log;
  error_log     /var/log/nginx/api.recsys.cpaier.com.error.log;

  ssl_certificate      /etc/letsencrypt/live/api.recsys.cpaier.com/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/api.recsys.cpaier.com/privkey.pem;

  location / {
    include uwsgi_params;
    uwsgi_pass recsysapi;
    uwsgi_param UWSGI_SCHEME https;
  }
}
```

Figure A.14.: Basic nginx config

Appendix B.

Appendix: User Study

```
def sus_score(items):
  score = 0
  for index in range(0, len(items)):
    pos = index + 1
    rating = int(items[index])

    if pos % 2 == 1:
      score += rating - 1
    else:
      score += 5 - rating
  return score * 2.5
```

Figure B.1.: Calculate SUS with python

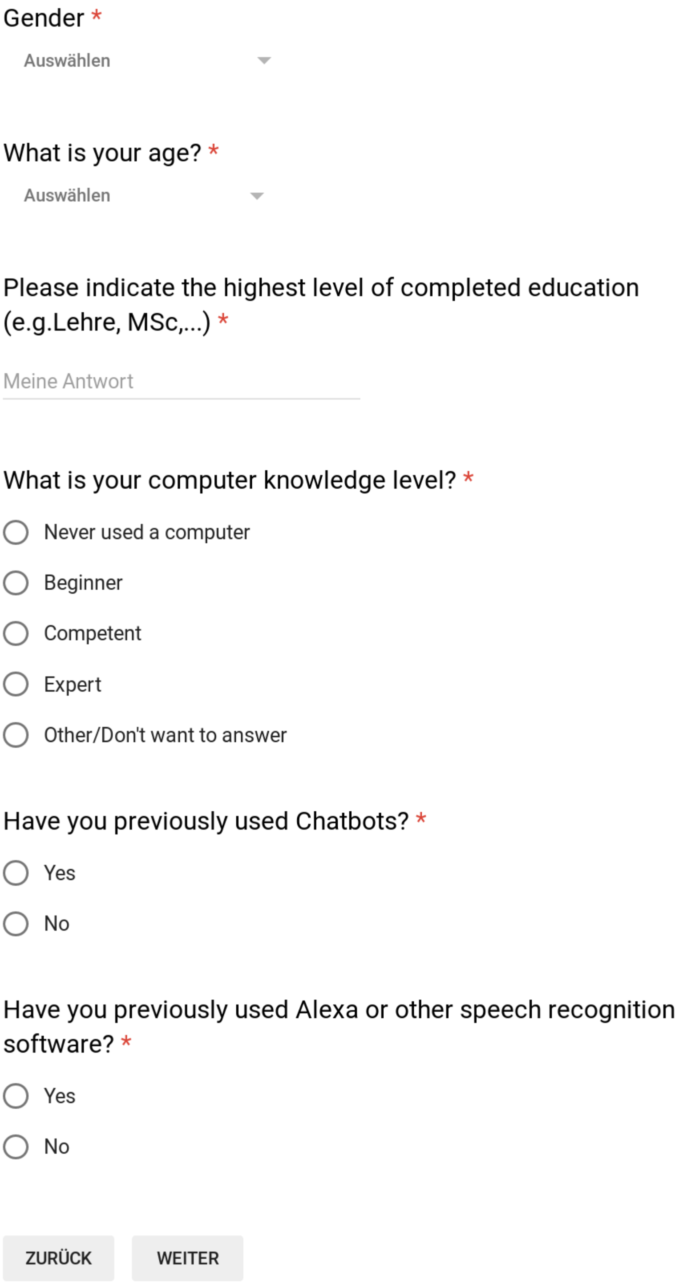
```
Thank you for participating in this user study. This will take about 30−45 minutes.

For my master thesis I want to compare various interfaces for recommender systems. Recommender systems are
used nowadays all over the Internet, for example at Spotify or Netflix to show you music or movies you
might like or at Amazon where items are (not only) recommended by your past shopping behaviour.

With this thesis, alternative approaches to use such systems should be investigated for their usability. You
are going to use a movie recommender on three ways: a webform (Task 1), a chatbot (task 2) and by using
voice commands with Alexa (Task 3). The quality of the recommended movies should not be evaluated, as those
are not part of this work. Finally I ask you to fill out a questionaire to get your feedback.  Please note
that any data that you explicitly put into any of the interfaces will be recorded together with a sesson
token. Beyond that, no data, and explicitly no sensitive data, is recieved or stored.

# TASKS For each interface please try the following tasks:
## TASK 1:
− Choose at least two movies and select them
− Click, if available, on the IMDB link on one of the movies
− Remove one of the selected movies
− Start the recommender and wait for the results

## TASK 2:
− Search and add a movie without the complete name (e.g. "dark" to get the movie "The Dark Knight")
− Add one or more genres
− Add at least one person by searching the exact name (e.g. "Brad Pitt")
− Start the recommender and wait for the results

## Task 3:
− Add any movie, genre or person you like
− Add any movie, genre or person you do not like
− Add a keyword
− Start the recommender and wait for the results

## Task 4:
− Add any item you like
− Restrict your search by selecting a year or a time frame
− Mark at least one item as undesired
− Start the recommender and wait for the results
− Upvote or downvote at least one movie

#### 1. WEBFORM
Visit https://recsys.cpaier.com with any browser like Google Chrome or Mozilla Firefox. The webform that
you see is structured in three parts: first the search area where you can search an item and where you can
pick your movies from, the centered panel with the already selected movies and the third panel that displays
the recommended movies. You can choose whether you like or dislike a movie in the second panel and in the
last panel you can up or downvote the results. Additionally each movie has a link to its IMDB entry.

#### 2. CHATBOT
You can either do this task with Telegram Messenger or Facebook. Each interface leads to the same result. The
chatbot is capable of having some kind of conversation with you. You can start by telling what movie you like
or do not like or you might simply say "hi". Be aware that the chatbot is only capable of understanding
simple sentences. If you are lost you can always use the command "help" or "help all" to get an overview
of its capabilities.

###### SETUP INSTRUCTIONS TELEGRAM
Visit the URL http://telegram.me/recsys_cp_bot with your telegram enabled device or search for
recsys_cp_bot and add it to your contacts. Press the "Start" button below to start the conversation.

###### SETUP INSTRUCTIONS FACEBOOK
Come back to me for login credentials, visit https://m.me/174771719834215 and start a
conversation with this bot

#### 3. VOICE COMMANDS
You can try "Alexa, movie recommender" to state that you want to use the recommender interface. Please be
aware that stating that command you are in the movie recommender until the LED is going of. If you have
voluntarily (for example, by using the command "Alexa stop") or unvoluntarily ended the session you can
start right where your last one ended. You can also append one of the commands below to the "Alexa, [tell the]
movie recommender" as well.  You can tell Alexa if you like or hate a movie, genre, person, keyword or year
e.g. by stating "I like the movie Skyfall" or "I like crime". If there is a list of multiple results you
can say "more" to get more results or previous to jump one page back. You can use the select, remove, up and
downvote functions by mentioning the number of the movie, like for the chatbot. For example "Select one" will
select the first movie of the search results. Finally the command "recommend" will calculate the recommended
items. You can ask for the result by saying "Results", or "Do you have results?"  By the way: in most cases
Alexa is not capable of listening to you while talking, so wait until it's your turn!

#### SURVEY
Please fill out the survey at: https://goo.gl/aMQkE7
```

Figure B.2.: Invitation to participate in the user study, Alexa version

Appendix B. Appendix: User Study

```
Thank you for participating in this user study. This will take about 30-45 minutes.

For my master thesis I want to compare various interfaces for recommender systems. Recommender systems are
used nowadays all over the Internet, for example at Spotify or Netflix to show you music or movies you
might like or at Amazon where items are (not only) recommended by your past shopping behaviour.

With this thesis, alternative approaches to use such systems should be investigated for their usability. You
are going to use a movie recommender on two ways: a webform (Task 1), a chatbot (task 2).
The quality of the recommended movies should not be evaluated, as those
are not part of this work. Finally I ask you to fill out a questionaire to get your feedback.  Please note
that any data that you explicitly put into any of the interfaces will be recorded together with a sesson
token. Beyond that, no data, and explicitly no sensitive data, is recieved or stored.

# TASKS For each interface please try the following tasks:
## TASK 1:
- Choose at least two movies and select them
- Click, if available, on the IMDB link on one of the movies
- Remove one of the selected movies
- Start the recommender and wait for the results

## TASK 2:
- Search and add a movie without the complete name (e.g. "dark" to get the movie "The Dark Knight")
- Add one or more genres
- Add at least one person by searching the exact name (e.g. "Brad Pitt")
- Start the recommender and wait for the results

## Task 3:
- Add any movie, genre or person you like
- Add any movie, genre or person you do not like
- Add a keyword
- Start the recommender and wait for the results

## Task 4:
- Add any item you like
- Restrict your search by selecting a year or a time frame
- Mark at least one item as undesired
- Start the recommender and wait for the results
- Upvote or downvote at least one movie

#### 1. WEBFORM
Visit https://recsys.cpaier.com with any browser like Google Chrome or Mozilla Firefox. The webform that
you see is structured in three parts: first the search area where you can search an item and where you can
pick your movies from, the centered panel with the already selected movies and the third panel that displays
the recommended movies. You can choose whether you like or dislike a movie in the second panel and in the
last panel you can up or downvote the results. Additionally each movie has a link to its IMDB entry.

#### 2. CHATBOT

#### SURVEY
Please fill out the survey at: https://goo.gl/mZgsrV
```

Figure B.3.: Invitation to participate in the user study, Chatbot version



Gender *

Auswählen

What is your age? *

Auswählen

Please indicate the highest level of completed education (e.g.Lehre, MSc,...) *

Meine Antwort

What is your computer knowledge level? *

○ Never used a computer
○ Beginner
○ Competent
○ Expert
○ Other/Don't want to answer

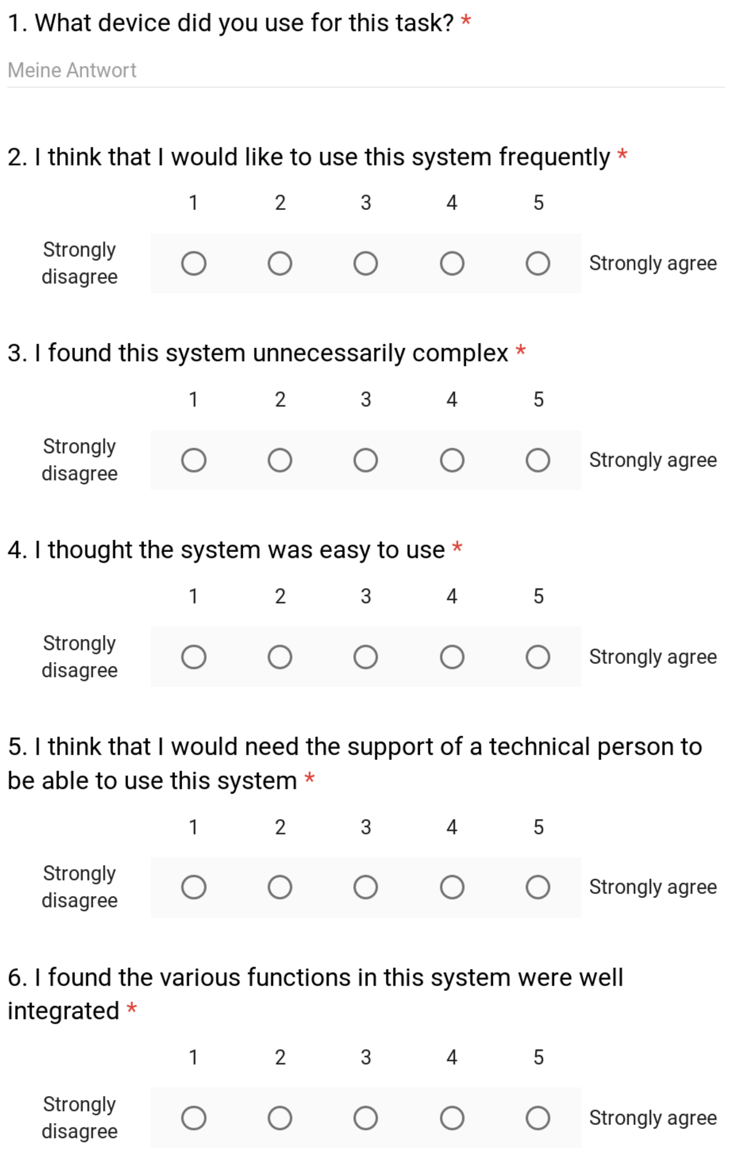
Have you previously used Chatbots? *

○ Yes
○ No

Have you previously used Alexa or other speech recognition software? *

○ Yes
○ No

Figure B.4.: Survey First Page



1. What device did you use for this task? *

Meine Antwort

2. I think that I would like to use this system frequently *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

3. I found this system unnecessarily complex *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

4. I thought the system was easy to use *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

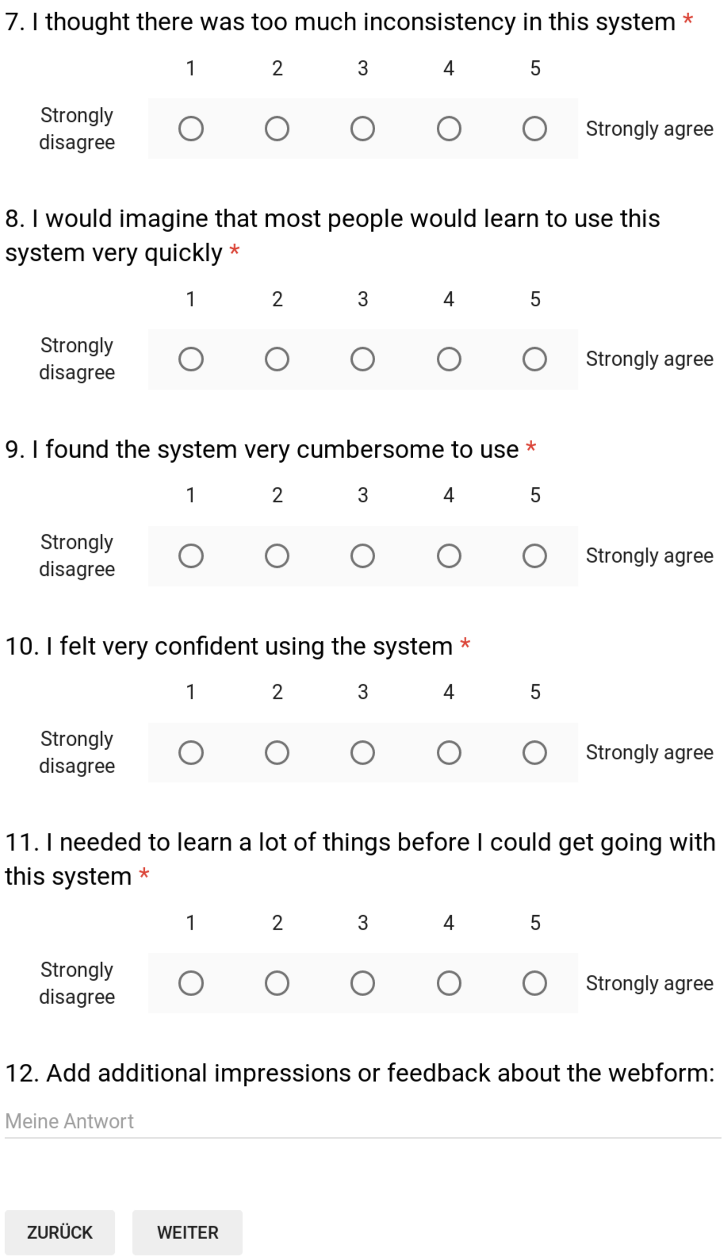
5. I think that I would need the support of a technical person to be able to use this system *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

6. I found the various functions in this system were well integrated *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

Figure B.5.: Survey Interface Page



7. I thought there was too much inconsistency in this system *

1 2 3 4 5

Strongly disagree ○ ○ ○ ○ ○ Strongly agree

8. I would imagine that most people would learn to use this system very quickly *

1 2 3 4 5

Strongly disagree ○ ○ ○ ○ ○ Strongly agree

9. I found the system very cumbersome to use *

1 2 3 4 5

Strongly disagree ○ ○ ○ ○ ○ Strongly agree

10. I felt very confident using the system *

1 2 3 4 5

Strongly disagree ○ ○ ○ ○ ○ Strongly agree

11. I needed to learn a lot of things before I could get going with this system *

1 2 3 4 5

Strongly disagree ○ ○ ○ ○ ○ Strongly agree

12. Add additional impressions or feedback about the webform:

Figure B.6.: Survey Interface Page (cont)

Bibliography

[1] *'I'm the original voice of Siri'*. URL: https://edition.cnn.com/2013/10/04/tech/mobile/bennett-siri-iphone-voice (visited on 09/03/2018) (cit. on p. 30).

[2] *5 SECOND USABILITY TESTS*. URL: https://measuringu.com/five-second-tests/ (visited on 04/24/2018) (cit. on p. 81).

[3] *A short Story about React Native*. URL: https://jobninja.com/blog/short-story-react-native/ (visited on 04/06/2018) (cit. on p. 40).

[4] *A.L.I.C.E. and AIML Documentation*. URL: https://sourceforge.net/projects/alicebot/files/Documentation/Don_t%20Read%20Me/dont.html/download (visited on 08/15/2018) (cit. on p. 19).

[5] *About the Jabberwacky AI*. URL: http://www.jabberwacky.com/j2about (visited on 08/15/2018) (cit. on p. 20).

[6] *Actions on Google*. URL: https://developers.google.com/actions/extending-the-assistant (visited on 09/03/2018) (cit. on p. 30).

[7] *AIML V2 Working Draft*. URL: https://docs.google.com/document/d/1wNT25hJRyupcG51aO89UcQEiG-HkXRXusukADpFnDs4/pub (visited on 08/12/2018) (cit. on p. 19).

[8] *Alexa Feature Help*. URL: https://www.amazon.com/gp/help/customer/display.html?nodeId=G201952240 (visited on 09/03/2018) (cit. on p. 29).

[9] *Amazon Echo Is A £199 Connected Speaker Packing An Always-On Siri-Style Assistant*. URL: https://techcrunch.com/2014/11/06/amazon-echo/ (visited on 09/03/2018) (cit. on p. 29).

[10] *Amazon explains how Alexa recorded a private conversation and sent it to another user*. URL: `https://www.theverge.com/2018/5/24/17391898/amazon-alexa-private-conversation-recording-explanation` (visited on 09/04/2018) (cit. on p. 30).

[11] *API.AI is joining Google!* URL: `https://blog.dialogflow.com/post/joining-google/` (visited on 08/26/2018) (cit. on p. 25).

[12] *Apple will release its £349 HomePod speaker on February 9th*. URL: `https://www.theverge.com/2018/1/23/16922682/apple-homepod-release-date-announced-preorders` (visited on 09/03/2018) (cit. on p. 29).

[13] Robert L. Armstrong. "The Midpoint on a Five-Point Likert-Type Scale." In: *Perceptual and Motor Skills* 64.2 (1987), pp. 359–362. DOI: `10.2466/pms.1987.64.2.359`. URL: `https://doi.org/10.2466/pms.1987.64.2.359` (cit. on p. 78).

[14] Aaron Bangor, Philip Kortum, and James Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale." In: *J. Usability Studies* 4.3 (May 2009), pp. 114–123. ISSN: 1931-3357. URL: `http://dl.acm.org/citation.cfm?id=2835587.2835589` (cit. on p. 80).

[15] Jordi Mongay Batalla and Maria Ledzińska. "ON REDUCING THE DETRIMENTAL INFORMATION FLOOD IN THE USE OF INTERNET." In: *Problems of Education in the 21st Century* 28 (Apr. 2011). ISSN: 1822-7864. URL: `http://journals.indexcopernicus.com/abstract.php?icid=939901` (cit. on p. 2).

[16] *Bellkor's pragmatic chaos wins 1 million Netflix prize by mere minutes*. URL: `https://www.wired.com/2009/09/bellkors-pragmatic-chaos-wins-1-million-netflix-prize/` (visited on 04/21/2018) (cit. on p. 10).

[17] John Brooke. *"SUS-A quick and dirty usability scale." Usability evaluation in industry*. ISBN: 9780748404605. CRC Press, June 1996. URL: `https://www.crcpress.com/product/isbn/9780748404605` (cit. on pp. 77, 78).

[18] John Brooke. “SUS: A Retrospective.” In: *J. Usability Studies* 8.2 (Feb. 2013), pp. 29–40. ISSN: 1931-3357. URL: http://dl.acm.org/citation.cfm?id=2817912.2817913 (cit. on pp. 78, 79, 81).

[19] *Build Skills with the Alexa Skills Kit*. URL: https://developer.amazon.com/docs/ask-overviews/build-skills-with-the-alexa-skills-kit.html (visited on 09/03/2018) (cit. on p. 29).

[20] *Build Skills with the Alexa Skills Kit*. URL: https://developer.amazon.com/docs/ask-overviews/build-skills-with-the-alexa-skills-kit.html (visited on 04/16/2018) (cit. on p. 66).

[21] Robin Burke. “Hybrid Recommender Systems: Survey and Experiments.” In: *User Modeling and User-Adapted Interaction* 12.4 (Nov. 2002), pp. 331–370. ISSN: 1573-1391. DOI: 10.1023/A:1021240730564. URL: https://doi.org/10.1023/A:1021240730564 (cit. on p. 7).

[22] Rollo Carpenter and Jonathan Freeman. “Computing machinery and the individual: the personal Turing test.” In: (July 2005), p. 4. URL: http://www.jabberwacky.com/s/PTT100605.pdf (cit. on p. 22).

[23] *ChatScript README Github Repository*. URL: https://github.com/bwilcox-1234/ChatScript (visited on 08/08/2018) (cit. on p. 61).

[24] *Cleverbot Data for Machine Learning*. URL: https://www.existor.com/products/cleverbot-data-for-machine-learning/ (visited on 08/15/2018) (cit. on pp. 20, 22).

[25] *Cleverbot data for machine learning*. URL: https://www.existor.com/products/cleverbot-data-for-machine-learning/ (visited on 08/20/2018) (cit. on pp. 27, 28).

[26] Kenneth Mark Colby, Sylvia Weber, and Franklin Dennis Hilf. “Artificial Paranoia.” In: *Artif. Intell.* 2.1 (Jan. 1971), pp. 1–25. ISSN: 0004-3702. DOI: 10.1016/0004-3702(71)90002-6. URL: http://dx.doi.org/10.1016/0004-3702(71)90002-6 (cit. on pp. 17, 18).

[27] *Cortana's got skills*. URL: https://developer.microsoft.com/en-us/cortana (visited on 09/03/2018) (cit. on p. 30).

[28] *Creating a chatbot with cognitive technologies*. URL: https://developer.ibm.com/code/topics/chatbot/ (visited on 08/27/2018) (cit. on p. 24).

[29] John Dawes. "Do Data Characteristics Change According to the Number of Scale Points Used? An Experiment Using 5-Point, 7-Point and 10-Point Scales." In: *International Journal of Market Research* 50.1 (Jan. 2008), pp. 61–104. DOI: 10.1177/147078530805000106. URL: http://dx.doi.org/10.1177/147078530805000106 (cit. on p. 78).

[30] *Definition of chatbot in English*. URL: https://en.oxforddictionaries.com/definition/chatbot (visited on 08/10/2018) (cit. on p. 15).

[31] *Dialogflow Documentation*. URL: https://dialogflow.com/docs (visited on 08/26/2018) (cit. on p. 25).

[32] *Dialogflow Documentation Machine Learning*. URL: https://dialogflow.com/docs/machine-learning (visited on 08/26/2018) (cit. on p. 25).

[33] *Dialogs with colorful personalities of early ai*. URL: https://web.stanford.edu/group/SHR/4-2/text/dialogues.html (visited on 08/14/2018) (cit. on p. 18).

[34] *ELIZA chatbot online*. URL: https://www.masswerk.at/elizabot/ (visited on 08/13/2018) (cit. on p. 17).

[35] *ES6 In Depth: Arrow functions*. URL: https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/ (visited on 04/07/2018) (cit. on p. 41).

[36] *Evolution of the web*. URL: http://www.evolutionoftheweb.com (visited on 08/07/2018) (cit. on p. 5).

[37] *Expanded Developer Preview of Notifications for Alexa*. URL: https://developer.amazon.com/blogs/alexa/post/833b9af4-26e6-47d2-a13d-bdbd9a257512/expanded-developer-preview-of-notifications-for-alexa-skills (visited on 04/16/2018) (cit. on p. 73).

[38] *Extending platform commonality through universal Windows apps*. URL: https://blogs.windows.com/buildingapps/2014/04/02/extending-platform-commonality-through-universal-windows-apps/ (visited on 09/03/2018) (cit. on p. 29).

[39] David A. Ferrucci et al. "Building Watson: An Overview of the DeepQA Project." In: *AI Magazine* 31.3 (2010), pp. 59–79. URL: http://dblp.uni-trier.de/db/journals/aim/aim31.html#FerrucciBCFGKLMNPSW10 (cit. on pp. 24, 26).

[40] *Firefly's dim light snuffed out*. URL: `https://www.wired.com/1999/08/fireflys-dim-light-snuffed-out/` (visited on 04/21/2018) (cit. on p. 10).

[41] Mark Gales and Steve Young. "The Application of Hidden Markov Models in Speech Recognition." In: *Found. Trends Signal Process.* 1.3 (Jan. 2007), pp. 195–304. ISSN: 1932-8346. DOI: `10.1561/2000000004`. URL: `http://dx.doi.org/10.1561/2000000004` (cit. on p. 31).

[42] Jesse James Garrett. *Ajax, A New Approach to Web Applications*. URL: `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/` (visited on 08/08/2018) (cit. on p. 6).

[43] Stefan Göbel et al., eds. *Technologies for Interactive Digital Storytelling and Entertainment. TIDSE 2003 Proceedings, March 24-26, 2003*. Vol. 9. Computer Graphik Edition. Stuttgart, Germany. Darmstadt, Germany: Fraunhofer IRB Verlag, 2003. URL: `http://www.verlag.fraunhofer.de/bookshop/artikel.jsp?v=217489&local=en` (cit. on p. 19).

[44] David Goldberg et al. "Using Collaborative Filtering to Weave an Information Tapestry." In: *Commun. ACM* 35.12 (Dec. 1992), pp. 61–70. ISSN: 0001-0782. DOI: `10.1145/138859.138867`. URL: `http://doi.acm.org/10.1145/138859.138867` (cit. on p. 9).

[45] *Google unveils Google Assistant, a virtual assistant that's a big upgrade to Google Now*. URL: `https://techcrunch.com/2016/05/18/google-unveils-google-assistant-a-big-upgrade-to-google-now/` (visited on 09/03/2018) (cit. on p. 30).

[46] Maria das Graças Bruno Marietto et al. "Artificial Intelligence MArkup Language: A Brief Tutorial." In: *CoRR* abs/1307.3091 (2013). arXiv: `1307.3091`. URL: `http://arxiv.org/abs/1307.3091` (cit. on p. 19).

[47] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. 1st. O'Reilly Media, Inc., 2014. ISBN: 9781449372620 (cit. on p. 47).

[48] *Home Page of The Loebner Prize in Artificial Intelligence*. URL: `https://web.archive.org/web/20140321191350/http://www.loebner.net:80/Prizef/loebner-prize.html` (visited on 08/24/2018) (cit. on pp. 16, 20, 22).

[49] *HTML 5, A vocabulary and associated APIs for HTML and XHTML*. URL: `https://www.w3.org/TR/html50/` (visited on 04/20/2018) (cit. on p. 6).

[50] *HTML 5.2, W3C Recommendation, 14 Dezember 2017*. URL: `https://www.w3.org/TR/html/` (visited on 04/20/2018) (cit. on p. 5).

[51] *IBM Opens Up Its Watson Cognitive Computer For Developers Everywhere*. URL: `https://www.forbes.com/sites/bruceupbin/2013/11/14/ibm-opens-up-watson-as-a-web-service/` (visited on 08/27/2018) (cit. on p. 24).

[52] *Information Management: A Proposal*. URL: `https://www.w3.org/History/1989/proposal.html` (visited on 04/20/2018) (cit. on p. 5).

[53] *Integrate Cortana today*. URL: `https://developer.microsoft.com/en-us/cortana/devices` (visited on 09/03/2018) (cit. on p. 30).

[54] *Interview with Steve Worswick*. URL: `http://aidreams.co.uk/forum/index.php?page=Steve_Worswick_Interview_-_Loebner_2013_winner` (visited on 08/17/2018) (cit. on p. 20).

[55] *Introducing Dialogflow, the name for API.AI*. URL: `https://blog.dialogflow.com/post/apiai-new-name-dialogflow-new-features/` (visited on 08/26/2018) (cit. on p. 25).

[56] *Jabberwacky Chatbot*. URL: `http://www.jabberwacky.com` (visited on 08/15/2018) (cit. on p. 22).

[57] Eric A. Jacobi Jennifer A; Benson. "Use of electronic shopping carts to generate personal recommendations." Patent US 6317722 (US). Nov. 2001. URL: `https://www.lens.org/lens/patent/US_6317722_B1` (cit. on p. 10).

[58] Dietmar Jannach et al. *Recommender Systems: An Introduction*. 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 9780521493369 (cit. on pp. 7, 8, 13, 15).

[59] *Jeopardy! The IBM Challenge (16.02.2011) Day 3*. URL: `https://www.youtube.com/watch?v=ByLbU0r574Q` (visited on 08/26/2018) (cit. on p. 24).

[60] Ben J. A. Kröse and Patrick P. van der Smagt. *An Introduction to Neural Networks*. Fourth. Amsterdam, The Netherlands: The University of Amsterdam, 1991 (cit. on p. 26).

[61] Neal Lathia, Stephen Hailes, and Licia Capra. "The Effect of Correlation Coefficients on Communities of Recommenders." In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. SAC '08. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 2000–2005. ISBN: 978-1-59593-753-7. DOI: `10.1145/1363686.1364172`. URL: `http://doi.acm.org/10.1145/1363686.1364172` (cit. on p. 11).

[62] R. Likert. *A Technique for the Measurement of Attitudes*. A Technique for the Measurement of Attitudes Nr. 136-165. publisher not identified, 1932. URL: `https://books.google.at/books?id=9rotAAAAYAAJ` (cit. on p. 78).

[63] *Loebner Prize*. URL: `https://www.aisb.org.uk/events/loebner-prize` (visited on 08/24/2018) (cit. on pp. 16, 20).

[64] Pattie Maes. "Agents That Reduce Work and Information Overload." In: *Commun. ACM* 37.7 (July 1994), pp. 30–40. ISSN: 0001-0782. DOI: `10.1145/176789.176792`. URL: `http://doi.acm.org/10.1145/176789.176792` (cit. on p. 2).

[65] *Marvin Minsky in comp.ai: Annual Minsky Loebner Prize Revocation Prize 1995 Announcement*. URL: `https://groups.google.com/d/msg/comp.ai/dZtU8vDD_bk/06d5alLQRo0J` (visited on 08/13/2018) (cit. on p. 16).

[66] Michael L. Mauldin. "ChatterBots, TinyMuds, and the Turing Test: Entering the Loebner Prize Competition." In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*. AAAI '94. Seattle, Washington, USA: American Association for Artificial Intelligence, 1994, pp. 16–21. ISBN: 0-262-61102-3. URL: `http://dl.acm.org/citation.cfm?id=199288.199285` (cit. on pp. 15, 17).

[67] *Messenger-Plattform Documentation*. URL: `https://developers.facebook.com/docs/messenger-platform` (visited on 04/16/2018) (cit. on p. 54).

[68] *Microsoft Joins Debate Over On-Line Privacy by Acquiring Firefly*. URL: http://www.nytimes.com/1998/04/10/business (visited on 04/21/2018) (cit. on p. 10).

[69] *Microsoft limits the Cortana search box in Windows 10 to Bing and Edge only*. URL: https://venturebeat.com/2016/04/28/microsoft-limits-the-cortana-search-box-in-windows-10-to-bing-and-edge-only/ (visited on 09/03/2018) (cit. on p. 30).

[70] *Microsoft's Cortana digital assistant guards user privacy with 'Notebook'*. URL: https://www.pcworld.com/article/2099943/microsofts-cortana-digital-assistant-guards-user-privacy-with-notebook.html (visited on 09/03/2018) (cit. on p. 30).

[71] K. Nath, S. Dhar, and S. Basishtha. "Web 1.0 to Web 3.0 - Evolution of the Web and its various challenges." In: *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*. Feb. 2014, pp. 86–89. DOI: 10.1109/ICROIT.2014.6798297 (cit. on pp. 6, 36).

[72] *Österreich Zahlen Daten Fakten*. 13th. Statistik Austria, 2018. ISBN: 978-3-903106-66-6. URL: http://www.statistik.at/web_de/services/oesterreich_zahlen_daten_fakten/index.html (visited on 04/25/2018) (cit. on p. 93).

[73] Christian Paier. "Cheating in Recommender Systems." Bachelor's Thesis. Graz University of Technology, 2017. URL: https://cpaier.com/2017-09-15_bakk_cpaier.pdf (cit. on pp. 7, 11, 12, 14).

[74] *Pandorabots: Frequently Asked Questions*. URL: https://pandorabots.com/docs/content/faq.html (visited on 08/17/2018) (cit. on pp. 20, 24).

[75] Ayse Pinar Saygin, Ilyas Cicekli, and Varol Akman. "Turing Test: 50 Years Later." In: *Minds and Machines* 10.4 (Nov. 2000), pp. 463–518. ISSN: 1572-8641. DOI: 10.1023/A:1011288000451. URL: https://doi.org/10.1023/A:1011288000451 (cit. on p. 16).

[76] Lawrence R. Rabiner. "Readings in Speech Recognition." In: ed. by Alex Waibel and Kai-Fu Lee. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. Chap. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pp. 267–

296. ISBN: 1-55860-124-4. URL: http://dl.acm.org/citation.cfm?id=108235.108253 (cit. on p. 31).

[77] Paul Resnick et al. "GroupLens: An Open Architecture for Collaborative Filtering of Netnews." In: *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*. CSCW '94. Chapel Hill, North Carolina, USA: ACM, 1994, pp. 175–186. ISBN: 0-89791-689-1. DOI: 10.1145/192844.192905. URL: http://doi.acm.org/10.1145/192844.192905 (cit. on p. 9).

[78] Francesco Ricci et al., eds. *Recommender Systems Handbook*. Springer US, 2011. DOI: 10.1007/978-0-387-85820-3. URL: https://doi.org/10.1007/978-0-387-85820-3 (cit. on pp. 7, 8, 13, 14).

[79] *RiveScript Tutorial*. URL: https://www.rivescript.com/docs/tutorial (visited on 04/16/2018) (cit. on p. 61).

[80] John R. Searle. "Minds, brains, and programs." In: *Behavioral and Brain Sciences* 3 (1980), pp. 417–424 (cit. on p. 16).

[81] *SIRI RISING: The Inside Story Of Siri's Origins — And Why She Could Overshadow The iPhone*. URL: https://www.huffingtonpost.com/2013/01/22/siri-do-engine-apple-iphone_n_2499165.html (visited on 09/03/2018) (cit. on p. 29).

[82] *Speech Synthesis Markup Language (SSML) Reference*. URL: https://developer.amazon.com/docs/custom-skills/speech-synthesis-markup-language-ssml-reference.html (visited on 04/16/2018) (cit. on p. 73).

[83] *The BellKor Solution to the Netflix Grand Prize*. URL: http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf (visited on 04/21/2018) (cit. on p. 10).

[84] *The Big Internet Brands Of The '90s — Where Are They Now?* URL: https://www.npr.org/sections/alltechconsidered/2016/07/25/487097344/the-big-internet-brands-of-the-90s-where-are-they-now (visited on 08/07/2018) (cit. on p. 6).

[85] *The BigChaos Solution to the Netflix Grand Prize*. URL: http://www.netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf (visited on 04/21/2018) (cit. on p. 10).

[86] *The first webpage*. URL: `http://info.cern.ch/hypertext/WWW/TheProject.html` (visited on 04/20/2018) (cit. on p. 5).

[87] *The Original HTTP as defined in 1991*. URL: `https://www.w3.org/Protocols/HTTP/AsImplemented.html` (visited on 04/20/2018) (cit. on p. 5).

[88] *The Pragmatic Theory solution to the Netflix Grand Prize*. URL: `http://netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf` (visited on 04/21/2018) (cit. on p. 10).

[89] *The WOrldWideWeb browser*. URL: `https://www.w3.org/People/Berners-Lee/WorldWideWeb.html` (visited on 04/26/2018) (cit. on p. 5).

[90] *Today in Apple history: Siri debuts on iPhone 4s*. URL: `https://www.cultofmac.com/447783/today-in-apple-history-siri-makes-its-public-debut-on-iphone-4s/` (visited on 09/03/2018) (cit. on p. 29).

[91] Thomas Tullis and Jacqueline N Stetson. "A Comparison of Questionnaires for Assessing Website Usability." In: (June 2006) (cit. on p. 81).

[92] Alan M. Turing. "Computing Machinery and Intelligence." In: *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*. Ed. by Robert Epstein, Gary Roberts, and Grace Beber. Dordrecht: Springer Netherlands, 2009, pp. 23–65. ISBN: 978-1-4020-6710-5. DOI: `10.1007/978-1-4020-6710-5_3`. URL: `https://doi.org/10.1007/978-1-4020-6710-5_3` (cit. on p. 1).

[93] *Usability.gov: Glossry*. URL: `https://www.usability.gov/what-and-why/glossary/u/index.html` (visited on 04/25/2018) (cit. on p. 77).

[94] Tom Vanderbilt. *The science behind the Netflix algorithms that decide what you'll watch next*. 2013. URL: `http://www.wired.com/2013/08/qq_netflix-%20algorithm/` (visited on 04/21/2018) (cit. on p. 8).

[95] Oriol Vinyals and Quoc V. Le. "A Neural Conversational Model." In: *CoRR* abs/1506.05869 (2015). arXiv: `1506.05869`. URL: `http://arxiv.org/abs/1506.05869` (cit. on pp. 26, 28).

[96] Richard Wallace. *The elements of AIML style. ALICE AI Foundation.* 2004 (cit. on p. 19).

[97] Richard S. Wallace. “The Anatomy of A.L.I.C.E.” In: *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*. Ed. by Robert Epstein, Gary Roberts, and Grace Beber. Dordrecht: Springer Netherlands, 2009, pp. 181–210. ISBN: 978-1-4020-6710-5. DOI: `10.1007/978-1-4020-6710-5_13`. URL: `https://doi.org/10.1007/978-1-4020-6710-5_13` (cit. on p. 19).

[98] Joseph Weizenbaum. *Computer Power and Human Reason: From Judgment to Calculation*. New York: W. H. Freeman & Co., 1976. ISBN: 0-7167-0463-3 (cit. on p. 17).

[99] Joseph Weizenbaum. “ELIZA&Mdash;a Computer Program for the Study of Natural Language Communication Between Man and Machine.” In: *Commun. ACM* 9.1 (Jan. 1966), pp. 36–45. ISSN: 0001-0782. DOI: `10.1145/365153.365168`. URL: `http://doi.acm.org/10.1145/365153.365168` (cit. on pp. 17, 19).

[100] *What happened when a professor built a chatbot to be his teaching assistant*. URL: `https://www.washingtonpost.com/news/innovations/wp/2016/05/11/this-professor-stunned-his-students-when-he-revealed-the-secret-identity-of-his-teaching-assistant/?noredirect=on&utm_term=.8f0c88ad7b5b` (visited on 08/26/2018) (cit. on p. 25).

[101] *What is SaaS?* URL: `https://azure.microsoft.com/en-us/overview/what-is-saas/` (visited on 08/28/2018) (cit. on p. 22).

[102] *What is Web 2.0*. URL: `http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html` (visited on 04/20/2018) (cit. on pp. 2, 6).

[103] *Works with Alexa Certification Program*. URL: `https://developer.amazon.com/alexa/smart-home/launch/works-with-alexa` (visited on 09/03/2018) (cit. on p. 29).

[104] M. Wu. “Collaborative Filtering via Ensembles of Matrix Factorizations.” In: *KDD Cup and Workshop 2007*. Max-Planck-Gesellschaft. Aug. 2007, pp. 43–47 (cit. on p. 13).