Chatbots: History, Technology, and a Case Analysis

Benjamin Jay

State University of New York at New Paltz
Abstract

This thesis examines the more than 50 year history of chatbots that led to the development of Amazon’s Alexa, Google’s Assistant, and Apple’s Siri. A chatbot, commonly known as a conversational agent, is a computer framework that can have a normal conversation with a user by using a natural language processor (Reshmi and Balakrishnan, 2018). The goal is to understand the psychological and mathematical theories that worked well throughout history, as well as those that did not, and the impact they had on the evolution of modern chatbots. This thesis incorporates these theories into a new chatbot created using Google’s chatbot AI platform called Dialogflow. By following a Coursera course titled Building Conversational Experiences with Dialogflow, this thesis creates a chatbot that can schedule tours of a school and can answer questions about the SUNY New Paltz 2020 Commencement ceremony. Creating even the most basic chatbot requires a comprehensive understanding of the underlying theories and extensive coding experience (Abdul-Kader & Woods, 2015). This thesis assumes a foundation knowledge of computer coding.

Key Words: AIML, Chatbot, Conversational Agent, Google Dialogflow, Loebner Test
# Table of Contents

Defining a chatbot ............................................................................................................. 4  
The creation of the first chatbot ...................................................................................... 4  
Loebner Test .................................................................................................................... 5  
ALICE and AIML .............................................................................................................. 6  
Modern trends .................................................................................................................. 12  
Creating a chatbot using Google Dialogflow ................................................................. 16  
Building a chatbot: a case analysis ............................................................................... 20  
Writing to a backend database ....................................................................................... 21  
Reading from a backend knowledge base ...................................................................... 23  
Conclusion ....................................................................................................................... 26  
References ....................................................................................................................... 27  
Appendix ......................................................................................................................... 31
Defining a chatbot

Chatbots are referred to as many other names, such as; conversational agents, human-computer dialogue systems, interactive agents, virtual agents, virtual humans, and virtual assistants (Abu Shawar & Atwell, 2005 and Palanica et al, 2019). While there may be minute differences between these terms, it is generally accepted that they can be used interchangeably. Chatbots blur the line between human and not human because they attempt to mimic human speech patterns (Abu Shawar & Atwell, 2005). The most basic understanding of chatbots is that they are frameworks that respond to a natural language input with a natural language-like output (Reshmi and Balakrishnan, 2018).

The creation of the first chatbot

The first chatbot was introduced in 1966 by an MIT scientist named Joseph Weizenbaum (Abu Shawar & Atwell, 2005). Weizenbaum named his chatbot ELIZA, though there are many conflicting reasons for his doing so (Bradeško & Mladenić, n.d.). Nonetheless, ELIZA was the first of its kind (Prezgalinska, 2019). ELIZA’s most popular script was DOCTOR, which simulated a Rogerian psychotherapist (Prezgalinska et al, 2019). ELIZA had poor conversational ability and could easily be stymied by complex questions (Bradeško & Mladenić, n.d.). However, patients were easily tricked by ELIZA since they had never experienced a natural language conversation with a non human (Bradeško & Mladenić, n.d.). Despite only being able to converse on an extremely basic level, most patients stated that they felt a connection to ELIZA (Prezgalinska et al, 2019). See appendix A for a sample conversation with ELIZA’s script DOCTOR and a link to start a conversation with ELIZA, recreated by Norbert Landsteiner in 2005.
The development of ELIZA led to many similar products in the years following its release (Kim, Cha, & Kim, 2019). In 1973, psychiatrist Kenneth Colby of Stanford University created a chatbot called PARRY (Abu Shawar & Atwell, 2005). PARRY was designed to simulate a paranoid patient suffering from schizophrenia (Abu Shawar & Atwell, 2005). To test PARRY’s effectiveness, psychiatrists were given a transcript from either a psychiatrist’s conversation with PARRY or a psychiatrist’s conversation with a real paranoid patient (Bradeško & Mladenić, n.d.). The psychiatrists were then asked to identify if the transcript was that of a conversation with PARRY or that of a conversation with an actual patient (Bradeško & Mladenić, n.d.). Psychiatrists correctly identified the conversation with PARRY approximately 48% of the time (Bradeško & Mladenić, n.d.). However, this is likely due to the psychiatrists’ lack of exposure to non-human machines capable of producing natural language, and is not meant as a testament to PARRY’s modern day effectiveness (Bradeško & Mladenić, n.d.).

Loebner test

This method of testing if PARRY could mimic real life conversational output was the first implementation of a Turing test — literally meaning imitation games — to a chatbot (Bradeško & Mladenić, n.d.). As chatbots began to rise in popularity, computer scientists developed the Loebner test as a type of Turing test to measure chatbots’ effectiveness (Abu Shawar & Atwell, 2017). In the case of the Loebner competition, judges speak to each chatbot for 10-15 minutes and rate each chatbot in terms of naturalness which is inherently a subjective term (Abu Shawar & Atwell, 2017). The annual competition is won by whichever chatbot is rated highest by the judges in terms of naturalness (Abu Shawar & Atwell, 2017). Huge strides in chatbot innovation have been made since the Loebner test was created in 1991, although no
chatbot has won the gold medal for tricking all 10 judges into thinking it was a natural conversation (Abdul-Kader & Woods, 2015). See appendix B for a list of each chatbot that has won the competition and what design techniques each chatbot uses, created by Bradeško and Mladenić (n.d.).

ALICE and AIML

ALICE was the first chatbot to take the world by storm since ELIZA. ALICE, which stands for artificial linguistic internet computer entity, was developed by Richard Wallace in 1995 (Reshmi and Balakrishnan, 2018). ALICE’s revolutionary technology won the Loebner prize for two consecutive years in 2000, 2001, and again in 2004 (Bradeško & Mladenić, n.d.). ALICE was revolutionary because it was the first chatbot that used Artificial Intelligence Markup Language (AIML), a derivative of Extensible Markup Language (XML) (Kim, Cha, & Kim, 2019). The most fundamental building blocks of AIML are optional topics and mandatory categories; all of which must be hand coded (Abu Shawar & Atwell, 2005). In one sense, this makes the possibility endless since the chatbot can learn to respond to any category that is hand coded into it (Abu Shawar & Atwell, 2015). However, this also largely restricts the limits of the chatbot because hand coding takes hours, and only so much can realistically be hand coded (Abu Shawar & Atwell, 2005). The first publicly available version of ALICE consisted of 50,000 different categories (Abu Shawar & Atwell, 2005).

The dialogue with a chatbot consists of two main parts: The natural language input and the template response (Wallace, 2003). In an AIML chatbot, the natural language input is translated into a pattern, which is then assigned to a category and mapped directly to a template response (Abu Shawar & Atwell, 2005). There are three types of categories that AIML offers:
1. Atomic categories: These provide 1:1 mapping, where a specific natural language input automatically maps to a precoded template response (Abu Shawar & Atwell, 2005). Here is example coding of an atomic category:

   `<category><pattern>HELLO ALICE</category></pattern>
   `<template>Hello</template>`

In this example, ALICE automatically recognizes the natural language input of “HELLO ALICE,” and maps it to a category that has a pre-generated response of “Hello”. This example was created by Abu Shawar & Atwell’s in their 2005 study.

2. Default categories: These involve wildcard symbols of * and _, which are default to the order of words (instead of using all the words themselves) (Abu Shawar & Atwell, 2005). Here is an example coding of a default category:

   `<category><pattern>HELLO ROBOT</pattern>
   `<template>Hello friend</template>`

In this example, ALICE could not find an atomic 1:1 matching of “HELLO ROBOT,” to a category. Therefore, it defaults to a category with the pattern of “Hello *” and replies with the template of “Hello friend.” This example is also from Shar & Atwell’s study in 2005.

3. Recursive categories: This special category uses the `<srai>` and `<sr>` tags. While the meaning of these acronyms has no consensus, many believe it stands for “syntactic rewrite” (Wallace 2003). This meaning is unclear and vague because it has several different uses. The
following codes are all excerpted from Wallace’s initial paper written in 2003 about his experience creating ALICE.

a. Symbolic reduction: Creates simpler grammatical structure given a complex input
   
   `<category> <pattern>DO YOU KNOW WHAT THE * IS</pattern> <template> <srai>What is *<star/></srai></template> </category>`

   In this example, ALICE reduces the complex user input into a simpler form before it maps the input pattern to a template response.

b. Divide and conquer: Splits an input into multiple different inputs and creates a combined response for each input.
   
   `<category>
   <pattern>YES *</pattern>
   <template><srai>YES</srai> <sr/></template>
   </category>`

   In this example, ALICE will treat “Yes” as one input and the following words as a completely separate input. This will pull two separate categories and generate two separate templates (although ALICE will combine both templates into one response).

c. Synonyms: Varies the template response so it uses different words, but has the same meaning.
   
   `<category>`
In this example, ALICE responds to the input of “Hi” with “Hello.” However, ALICE also could have answered with “Hey,” or “Howdy.”

d. Spelling and Grammar: Corrects spelling and grammar from the input pattern.

In this example, ALICE clarifies with the user which form of “your” is correct.

e. Key word: Generates the same template response anytime a keyword input is recognized.
In this example, ALICE recognizes the keyword input “MOTHER” and generates the same response “Tell me more about your family” for all the variations of the input.

f. Conditional: This requires ALICE to remember an object whose value was set earlier in the conversation.

In this example, ALICE retrieves the set antecedent for “he”, and if no value was set, then ALICE replies “I don’t know who he is.”
g. Targeting: ALICE learns new replies from its users.

User: Where do I find the web page with Australopithecus?

Alice: Where do you want to find it?

User: Bad Answer Alice, time to learn a new line.

Alice: How would you answer "Where do I find the web page with Australopithecus?"

User: I would say: "http://www.ancientskulls.net/australopithecus.html"

Alice: Thanks, human. Try asking me again.

User: Where do I find the web page with Australopithecus?

Alice: http://www.ancientskulls.net/australopithecus.html

h. Context: ALICE must remember what it said previously in order to know how to respond.

<category>

<pattern>YES</pattern>

<that>DO YOU LIKE MOVIES</that>

<template>What is your favorite movie?</template>

</category>

In this example, ALICE needs to recall what the user is responding “yes” to before it can ask a coherent follow up question.
After understanding how ALICE uses AIML to respond to natural language inputs, it is no wonder it revolutionized chatbots. However, ALICE’s main drawback is that all of its categories have to be hand coded (Abu Shawar & Atwell, 2005). The easiest way to fix this issue is by finding a way to automatically generate these categories. A study by Abu Shawar & Atwell (2005) uses a corpus based approach to generate categories. Abu Shawar converted corpus documents (otherwise known as readable texts) such as monologues, dialogues, and FAQ’s into AIML format. This generated the largest AIML based bot created with over 1,100,000 categories - an astronomical number in comparison to ALICE’s mere 50,000 categories. Abu Shawar’s chatbot effectively animated a corpus, further blurring the line between human and not human.

Jabberwacky is a chatbot created by Rollo Carpenter in 1997, two years after ALICE’s release (Kim, Cha, & Kim, 2019). Although Jabberwacky is not AIML based, it won the Loebner award in 2005 and 2006 (Bradeško & Mladenić, n.d.). Unlike ALICE, Jabberwacky has no hard coded, fixed categories or rules. Instead, Jabberwacky learns entirely from its previous user interactions (Kim, Cha, & Kim, 2019). See appendix C for a sample conversation with Jabberwacky and a link to start a conversation with Jabberwacky.

Modern Trends

Chatbots have gained extreme popularity since Jabberwacky and ALICE revolutionized the artificial intelligence industry. Since the early 2000s, there has been a trend towards more modern coding in chatbot creation. One major improvement is the development of ontology based chatbots (Abdul-Kader & Woods, 2015). Ontology creates a hierarchical structure of categories for a chatbot (Abdul-Kader & Woods, 2015). Let’s use a banking chatbot as an example. “Help,” may be the parent category and “accessing my account,” “depositing money,”
and “withdrawing money” may be the child categories. Going one step further, “I forgot my online ID,” “I forgot my online password,” “I do not know my account number,” may all be child categories of “accessing my account” (and therefore grandchild categories of the parent “Help”). Ontology also allows a chatbot to draw relationships between different hierarchical families. Here is a theoretical example of a dialogue from a banking chatbot that is ontology based:

User: I need help

Bot: Do you need help with withdrawing money, depositing money, accessing your account, or other?

User: I need help accessing my account

Bot: Did you forget your online ID, your password, or your account number?

User: I forgot my online ID

Bot: Please click the following link to have your online ID emailed to you: sample.link.com

Do you also need help checking your account balance?

User: Yes please

Bot: After you have your username, click the following link to view your account balance: secondsample.link.com

User: Thank you, Bot!

Bot: You’re welcome.

Throughout this exchange, the bot lists each child category until it pinpoints the user’s request. Then it draws a connection to another hierarchical family to predict what the user may
need help with after finding out its online ID. In general, using ontology allows for a more natural dialogue flow instead of a simple question and answer format (Hallili, 2014).

Another recent trend in chatbots is the implementation of the Markov chain model (Bradeško & Mladenič, n.d.). Mathematician AA Markov developed a probabilistic theory that was mathematical in nature (Basharin et al, 2004). This theory was coined as the Markov chain in 1926, just 4 years after AA Markov’s death (Basharin et al, 2004). Markov studied the sequence of 20,000 letters of the then-famous poem, “Eugine Onegin” (Basharin et al, 2004). Markov noted that the stationary probability of their being a vowel was p=.432, a vowel following another vowel was p=.128, and a vowel following a consonant was p=.663 (Basharin et al, 2004). Note that these later two values do not equal 100% likely due to words that begin with vowels.

The Markov chain was first used in a chatbot by a program called HeX, which won the Loebner prize in 1997 (Bradeško & Mladenič, n.d.). However, it has become increasingly popular throughout the last two decades. In a chatbot, the Markov chain analyzes the probability of a specific word following another word, instead of a letter following another letter (Bradeško & Mladenič, n.d.). Essentially, it uses fixed probability of the order of words to generate a response to the user (Bradeško & Mladenič, n.d.). Here is an example of determining which word will follow a set of two words:

“I love to eat chocolate ice cream on Wednesday afternoons.”

“I love” -> [“to”]

“love to” -> [“eat”]

“To eat” -> [“chocolate”]

“Eat chocolate” -> [“ice cream”]
A chatbot would analyze tens of thousands of these inputs to determine which word is most probabilistic to follow the previous set of two words. The Markov chain will continue to generate a response using this method until a set of two words is likely to be followed by a period. This process often results in a response that sounds correct, but makes no logical sense (Bradeško & Mladenić, n.d.).

The other trends include making chatbots easier to use, more accessible to use, and making their responses seem more natural and human-like. To make chatbots easier and more accessible, many have a speech recognition feature (Abdul-Kader & Woods, 2015). Using this feature, a user can say a request and the chatbot will use speech-to-text technology to convert the audio into a text input (Abdul-Kader & Woods, 2015). To make the chatbot responses more natural, some include human-like tendencies in the chatbot’s responses. These tendencies include non sequitur arguments, spelling and typing errors, and creating a personal backstory for the chatbot (Bradeško & Mladenić, n.d.). While these tendencies make the bot less formal, they increase the naturalness of a bot. The goal is to trick humans speaking with the bot that they are actually speaking with another human, just like in the Loebner test (Abu Shawar & Atwell, 2017).

Some of the most well known chatbots used in modern day include Microsoft’s Cortana, Amazon’s Alexa, Apple’s Siri, and Google’s Assistant (Sjöström et al, 2018). Chatbots are widely used across many different industries including banking (Przegalinska, 2019), higher education (Sjöström et al, 2018), ecommerce (Hallili, 2014), and healthcare (Palanika et al, 2019). These bots typically use the techniques explained above in combination with each other, instead of just picking one. In other words, many modern chatbots use some element of ontology, some element of the Markov chain, and some element of AIML (including recursive coding).
There are also many modern websites that allow a user to create and host their own chatbot. Some of the most famous chatbot creation programs include Motion.ai, Converse.ai, QnA Maker, Octane AI, IBM Watson, Botsify, Chatfuel, Pandorabots, Microsoft Bot Framework, Wit.ai, and Semantic Machine (Sjöström et al, 2018). Most of these programs support AIML code (Sjöström et al, 2018). However, creating a chatbot requires extensive programming skills in order to develop even a basic sense of realism (Abdul-Kader & Woods, 2015).

Creating a chatbot using Google Dialogflow

***note that the steps in this section are all from Coursera.com’s publicly available course titled Building Conversational Experiences with Dialogflow***

Creating a chatbot can be broken down into three broad phases: Design, development, and deployment. When designing a chatbot, it is important to define expected common user inputs and prioritize building a chatbot that can respond to them. A chatbot that does one thing well is more effective than a chatbot than does many things poorly. Designing a persona for a chatbot can also make the chatbot more human-like and allow for a more natural conversation with the intended user. A persona is a habitual external attitude of an individual person (Jolande, 2013) — or in this case, a chatbot. More specifically, a persona is a UX (user experience) concept that will trigger familiar conversational patterns used to make a chatbot more relatable to its intended user. On a very basic level, a persona can be extremely formal or informal. See appendix D for a list of formal versus informal language provided by Coursera.

Once the chatbot is designed, it’s time to begin developing, or building, the chatbot. The first step of building a chatbot is to define its intents. An intent is the action the user wants to
executive — this is typically the main verb in a user input (Chamchong & Wong, 2019). These are very similar to the categories used in ALICE or AIML coding. In an online banking chatbot, some examples include “Opening a bank account,” “Closing a bank account,” or “Updating terms of a bank account.” Intents must be mutually exclusive, meaning that there is no overlap between two different intents. This ensures that there is no ambiguity which intent a user input will be mapped to. It would be impossible to make intents collectively exhaustive, meaning a unique intent for every possible user input. To fix this problem, Dialogflow has an automatic fallback intent which a user input is mapped to if it does not match any other intent (Chamchong & Wong, 2019). By default, this will guarantee a user input is always mapped to an intent.

The next step is to add training phrases to each intent. These are examples of possible user inputs (Blaj & Balan, 2019). Dialogflow uses AI technology to learn from these training phrases and map similar inputs to the same intent. Continuing with the online banking chatbot example, using training phrases such as “I want to open a bank account,” “I need to open a bank account right now,” and “How can I open a bank account,” will all map to the same intent of “Opening a bank account.” Training phrases should vary in grammatical structure — active/passive voice, statement/question, generic/specific, formal/informal, etc.

Maintaining context allows the chatbot to keep track of where the user is in a conversation. It allows the chatbot to move seamlessly between intents without getting confused. Take the following exchange, for example:

User: I want to close my checking account with Bank of America based out of New York
Bot: No problem, your account will be closed in 4-5 business days. Would you like to buy a bond?
User: Yes please
By setting the output context of the intent “Close a bank account” as the same input context as “Upsell bonds,” the chatbot automatically tries to upsell a bond whenever a user closes a bank account. Similarly, the user replied “Yes” to buying a bond. Setting context for this affirmation eliminates the ambiguity of guessing to what the user meant by saying “Yes”. A context lasts for a preset amount of exchanges; the default number of exchanges is five, but this can be changed for each context.

After all intents are created with their respective training phrases, it is time to create entities. Entities extract useful facts in order to provide more information about an intent. Chamchong & Wong, 2019. For the online banking chatbot, a chatbot may have an entity labeled “Account type” with the possible values of “Savings,” “Checking,” or “CD.” Dialogflow allows for the values of entities to be automatically expanded throughout its conversations or for these values to be fixed. In this example, there are no other possible types of accounts, so the entity values should remain fixed. Entity values should not include stop words or prepositions such as “a checking account,” or “at Bank of America.” There are some default entities in Dialogflow such as “Time,” “Date,” “Address,” and “Name.” A composite entity can combine many different entities into one entity. An example of a composite entity would be “Account to create” which would be a combination of the entities “Account type,” “Name of bank,” and “Date.” Although there are some differences, on a very basic level, composite entities are a form of ontology.

Slot filling allows the chatbot to gather all the required entities before closing a specific intent. Slot filling can be envisioned as a mini context within an intent (instead of between intents). If a user’s initial input is missing a required entity, the chatbot will respond with follow
up questions until it has all the required entities to close an intent. Take the following exchange, for example:

User: I want to open an account.

Chatbot: Would you like to open a savings, checking, or CD account?

User: Checking account

Chatbot: What bank would you like to open an account with?

User: Bank of America

Chatbot: What is your legal name:

User: First name, Last Name

Chatbot: What is your email address?

User: example@email.com

Chatbot: Great! A request has been submitted. Please expect an email within 2-3 business days.

In order for the chatbot to close its “open an account” intent, slot filling was enabled to require four intents: the account type, bank name, user name, and user email address. The chatbot used pre-entered templates to ask the user for more information until it has all four required entities to close the intent.

Up until this point creating intents, training phrases, context, entities, and slot filling, was all drag and click using Google Dialogflow. However, storing this data requires hand coding using Dialogflow’s inline editor. The inline editor comes with prewritten code in node.js and a json file. In order to store this data, the chatbot will need to write entries into a backend database. The easiest backend database for Dialogflow to integrate with is Google Datastore, a serverless
NoSQL database (Creative Commons Attribution, 2020). Dialogflow communicates with Datastore through a webhook, which allows programs to send or receive data using APIs (Pérez-Soler et al, 2019). Fulfillment also needs to be enabled, which is the actual configuration of the external service (Pérez-Soler et al, 2019). In order to use Datastore, there must be a google datastore dependency added to the json file. In the node.js script, Google Datastore has to be imported with the correct project ID unique to the specific backend Datastore instance. There also must be a function for each intent that will be sent to Datastore, and the function must be mapped to the correct corresponding intent. A case analysis of adding a Datastore dependency, importing Datastore with the correct project ID, writing an intent function, and mapping that function to its intent will be provided in the following section.

Using a corpus based approach requires extracting entities from a back end knowledge base (or corpus) rather than writing into the backend database. Parsing through a corpus in Datastore requires creating a Google VM instance, a virtual server. This decreases the need for infrastructure, because Google hosts everything on the cloud. Working in VM instance requires intermediate knowledge of Python coding in cloud shell, especially when troubleshooting any issues. Generating the code to actually parse through the corpus would require extensive knowledge of javascript, so it is recommended to download pre existing code Google has. After successfully parsing through the corpus in the VM Instance and storing the entities in Datastore, a webhook needs to be set up in order to download those entities from Datastore to Dialogflow. This webhook code would also require extensive knowledge of javascripting, so it is recommended to use Angular-59 code (a front end app publicly available on Github). A case analysis of creating a corpus based chatbot will be provided in the next section.
Building a chatbot: a case analysis

The original design of this chatbot was an all-inclusive model that could answer any and all questions regarding the State University of New York at New Paltz. However, it quickly became clear that this would be too coding intensive for the scope of this thesis. Therefore, the chatbot focuses on two main components that it does well, rather than doing everything poorly. The first component allows the chatbot to write to a back-end database which stores entities that a user gives the chatbot. This component allows users to schedule tours at the college. The second component is a corpus based chatbot that parses through a back-end database to generate entities for Dialogflow. This chatbot can answer questions regarding the school’s 2020 commencement ceremony.

Writing to a backend database

***note that this case analysis uses Google Dialogflow and follows Coursera.com’s publicly available course titled Building Conversational Experiences with Dialogflow***

The first step to creating a chatbot that allows users to book tours is to create the intents, training phrases, and entities. This intent is simply labeled “tours.” The default response is added to ask the user if they want a handicap accessible tour. This was accomplished by adding an affirmative intent and negative intent that has the same input context as the “tours” intent output context. Training phrases include, but are not limited to:

- I need to schedule a tour for me and my friend.
- Can I bring my school to campus next week?
- I would like to schedule a campus tour for Monday at 5pm.
View a full list of training phrases in appendix E. The next step is to define the entities - what facts does the chatbot need to know to book the tour? Four entities are used:

- **Tour type**: A hand coded entity with two possible values; group or individual
- **Number of people**: A system generated entity that's value can be any integer number
- **Date**: A system generated entity that’s value is the date of the scheduled tour
- **Time**: A system generated entity that’s value is the time of the scheduled tour

Slot filling is enabled which makes these entities required. Then these entities are mapped to specific words in the training phrases. This mapping can also be viewed in appendix E via color coding. The inline editor must be enabled in order to successfully setup the webhook. After enabling the inline editor, the datastore dependency must be added to the json file by inputting the following code:

```
"dependencies": { "@google-cloud/datastore": "^0.8.0",}
```

After the dependency is configured, the actual “tours” function has to be added using the node.js coding. This function defines the entities that will be written to Datastore for the “intent” function, and creates a default confirmation response template for the template once a tour is scheduled. See appendix F for the full function that was added. Note that functions are prewritten for the default welcome and default fallback intents. Lastly, an intent map needs to be added in order to tie the tour function to the tour intent. This can be accomplished by adding the following line of code in node.js:

```javascript
let intentMap = new Map();

intentMap.set('Tours', Tours);

agent.handleRequest(intentMap);
```
Finally, the chatbot is created once the inline editor code is deployed. The following is a sample conversation that allows the bot to schedule a tour in Datastore.

User: Hi
Chatbot: Hello! How can I help you?
User: I’d like to schedule a tour
Chatbot: What date would you like to visit campus?
User: July 8th
Chatbot: What time would you like to book your tour?
User: 2:00pm
Chatbot: How many people are coming on the tour?
User: 3 people
Chatbot: Is this a group tour (6 or more people)?
User: No, just an individual tour
Chatbot: All done, do you want a handicap accessible tour?
User: No

This is the conclusion of a conversation where a user schedules a tour. See appendix G for an example of the information stored in Datastore. The second component involves creating a corpus based intent (and entities), which can be more complex.

Reading from a backend knowledge base

***note that this case analysis uses Google Dialogflow and follows Coursera.com’s publicly available course titled Building Conversational Experiences with Dialogflow***
The first step in creating a corpus based chatbot is to enable specific APIs on the Google Cloud Platform for Dialogflow, Cloud Storage, Cloud Source Repositories, and Cloud Natural Language. Enabling these APIs allows users to pull or write data from software offered by Google. The next step is to start a VM instance by initializing the app engine (part of the Google Cloud Platform). The App Engine uses python coding, not a simple click and drag UX experience. The directory, time zone, and name of the VM instance must be set. After the VM instance is started, the port must be changed to 8081, which is automatically compatible with Dialogflow.

For the purpose of this thesis, the chatbot uses pregenerated notebooks of code publicly available in a Cloud Storage bucket. In order to copy these notebooks from a Cloud Storage bucket, a gsutil command with the corresponding URL must be entered into a new notebook. After the notebooks successfully copy over from the Cloud Storage bucket, the corpus knowledge base must be uploaded. The commencement document was taken from the SUNY New Paltz website and converted into a text document. While no longer publicly available, see appendix H for a sample portion of the document.

After the corpus knowledge base is uploaded, the Process Handbook notebook (copied from the Cloud Storage bucket) is run. This notebook parses through the corpus knowledge base to extract the entity values into Datastore. By default, this notebook identifies a topic (or sample user input) as any line with less than five words. This notebook then assigns the topic a corresponding definition (or sample chatbot template output), which is any text following a topic but before a blank line. Therefore, in order to use the code from this Cloud Storage bucket, the corpus must be in the format of: the topic on the first line, the definition on the next line, then a blank line, then repeat.
This Process Handbook notebook will then extract these topics and definitions and store them in Datastore, under an entity called “Topic”. See appendix I for an example of how these topics and definitions appear in Datastore. Next, the Synonym Handbook notebook will automatically generate synonyms for all of the topics and store them as another entity in Datastore called “synonyms”. This helps create a more natural experience with the user, and allow the chatbot to respond to a variety of questions.

In order to copy these topics and definitions from Datastore into Dialogflow, there needs to be an entity in chatbot called “Topic” (which matches the name of the entity created in Datastore). After running the Dialogflow notebook, all the topics and definitions will be copied into the “Topic” entity in Dialogflow. The next step is to create an intent also named “Topic,” and add a lookup action with the entity name @Topic. This lookup action is what allows the chatbot to use the webhook to get the definitions for each topic. Under the fulfillment for this intent, webhook call needs to be enabled. After the webhook is enabled, training phrases need to be created for this intent and the entity values need to be mapped back to the “Topic” entity - see appendix J for the same training phrases used in the Commencement chatbot.

The last step is to actually deploy the webhook. This would take months to hand code, so instead the chatbot created for this thesis used Angular-59 code publicly available on Github - it was written by Jeff Delaney and licensed under the MIT license. After cloning this code by using the Google Cloud Shell, an environments folder needs to be created under this newly added Angular-59 directory. Under this environments folder, environment.ts and environment.prod.ts files needed to be added with the corresponding client access token from the Dialogflow. After running a npm install command and installing CLI globally, a “ngm build -prod” command will create a dist folder (which functions as the build folder) in the 59-Angular directory. Lastly,
creating an app.yaml file (a configuration file that puts text in key value format) inside the Angular directory will allow the “gcloud app deploy” to deploy the webhook, allowing the chatbot to fully function.

Here is a sample exchange conversion between a user and the chatbot:

User: Hi bot!

Chatbot: Greetings! How can I assist?

User: Will I receive my diploma at commencement?

Chatbot: No. The official diploma is not presented at Commencement. Diplomas will be mailed to the candidate’s permanent address approximately six-to-eight weeks after Commencement from the Records and Registration office. A diploma cannot be issued until all obligations, such as library fees, parking tickets, etc., are cleared. Note: When you cross the stage, you will be given a diploma cover which contains your Certification of Participation.

Conclusion

Chatbots are based on complex psychological, mathematical, and coding based theories. AIML’s use of atomic, default, and recursive categories are the backbone of most modern chatbots. Other fundamental theories include natural language processing, ontology, the markov chain, speech-to-text recognition. There are many platforms that allow users to create their own chatbots by using a combination of these theories. Following a Coursera course titled Building Conversational Experience with Dialogflow, this thesis walks through the steps needed to create a chatbot using Google’s Dialogflow AI platform. The chatbot created in this thesis uses two models; One that allows the chatbot to write into a backend database to schedule the user tours, and another that allows the chatbot to read from a backend database to answer user inquiries.
regarding SUNY New Paltz’s 2020 commencement. The findings of this thesis support previous studies that suggest: (1) Creating even a basic chatbot requires extensive coding experience and (2) a chatbot that does one or two things well is better than a chatbot that does a lot of things poorly.
References


Hallili, A. (2014, August 11). Toward an Ontology-Based Chatbot Endowed with Natural Language Processing and Generation. Retrieved July 18, 2020, from hal.inria.fr website: https://hal.inria.fr/hal-01089102


https://www.researchgate.net/publication/328245964_Designing_Chatbots_for_Higher_Education_Practice

Appendix A

This is a sample conversation with ELIZA’s most famous script, DOCTOR, which simulates a Rogerian psychotherapist. This implementation was designed by Robert Landsteiner in 2005. To start your own conversation with ELIZA, click the following link:

https://www.masswerk.at/elizabot/eliza.html
### Appendix B

<table>
<thead>
<tr>
<th>Year</th>
<th>Chatbot</th>
<th>Technology</th>
<th>Language Tricks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991</td>
<td>PC Therapist III (Weintraub, 1986)*</td>
<td>parsing, pattern matching, word vocabulary, remembers sentences</td>
<td>non sequitur, canned responses</td>
</tr>
<tr>
<td>1992</td>
<td>PC Professor (Weintraub, 1986)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>PC Politician (Weintraub, 1986)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td>TIPS (Whalen, 1994; Hutchens, 1997)</td>
<td>Pattern matching, database like system</td>
<td>Model of personal history</td>
</tr>
<tr>
<td>1995</td>
<td>PC Therapist (Weintraub, 1986)*</td>
<td>Same as in 1991</td>
<td>Same as in 1991</td>
</tr>
<tr>
<td>1996</td>
<td>HeX (Hutchens, 1997)</td>
<td>Pattern matching, Markov chain models to construct some replies</td>
<td>database of trick sentences, Model of personal history, not repeating itself</td>
</tr>
<tr>
<td></td>
<td></td>
<td>synonyms, list of proper names, ontology, database for storing facts</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>Albert One (Garner, 1995)</td>
<td>Pattern matching, hierarchical composition of other chat bots (Eliza, Fred, Sextalk)</td>
<td>Proactive monologues</td>
</tr>
<tr>
<td>1999</td>
<td>Albert One (Garner, 1995)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td>Ella (Copple, 2009)</td>
<td>Pattern matching, phrase normalization, abbreviation expansion, WordNet</td>
<td>Monologues, not repeating itself, identify gibberish, play knock-knock jokes</td>
</tr>
<tr>
<td>2006</td>
<td>Joan (Carpenter, 2006)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>UltrHAL by Robert Medeksza*</td>
<td>Combination of VB code and pattern matching scripts</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Elbot (Roberts, 2007)*</td>
<td>Commercial NLI system</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>Do-Much-More (Levy, 2009)*</td>
<td>Commercial property of Intelligent Toys Ltd.</td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Rosette (Wilcox, 2011)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table shows all Loebner prize winning Chatbots before 2013 with their relevant technologies. The table was created by Bradeško & Mladenić (n.d.). According to Bradeško & Mladenić (n.d.), “the winners marked with asterisk (*) are commercial programs and thus their technologies and internal structure is not publicly available.”
Appendix C

This is a sample conversion with the Jabberwacky chatbot. To start your own conversation with Jabberwacky, click the following link: [http://www.jabberwacky.com/](http://www.jabberwacky.com/)
Appendix D

This is a chart provided by Coursera, publicly available in their course titled *Building Conversational Experiences with Dialogflow*. On the left hand side are more formal terms while on the right hand side are more conversational terms.
Appendix E

<table>
<thead>
<tr>
<th>Training Phrase</th>
<th>Parameter Name</th>
<th>Entity</th>
<th>Resolved Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I want to schedule a tour for me and my friend</td>
<td></td>
<td>@Tour_Type</td>
<td>group</td>
</tr>
<tr>
<td>Can my parents and I come see campus</td>
<td></td>
<td>@sys.number</td>
<td>three</td>
</tr>
<tr>
<td>Schedule a group tour to visit campus</td>
<td></td>
<td>@sys.date</td>
<td>on Wednesday</td>
</tr>
<tr>
<td>Can I bring a group of three people to see the campus on a tour on Wednesday?</td>
<td></td>
<td>@Tour_Type</td>
<td>group</td>
</tr>
<tr>
<td>I need to see SUNY New Paltz on February 22nd</td>
<td></td>
<td>@sys.date</td>
<td>on February 22nd</td>
</tr>
<tr>
<td>Can I bring my school to campus next week?</td>
<td></td>
<td>@sys.date</td>
<td>next week</td>
</tr>
<tr>
<td>When do you offer tours of campus?</td>
<td></td>
<td>@sys.date</td>
<td>offer</td>
</tr>
<tr>
<td>When can I come visit campus?</td>
<td></td>
<td>@sys.date</td>
<td>visit</td>
</tr>
<tr>
<td>I need to schedule a tour</td>
<td></td>
<td>@sys.date</td>
<td>schedule</td>
</tr>
<tr>
<td>I would like to schedule a campus tour for Monday at 5pm</td>
<td></td>
<td>@sys.date</td>
<td>Monday at 5pm</td>
</tr>
</tbody>
</table>

These are the training phrases used in the chatbot created to schedule tours by writing to a backend data base. These training phrases are for the intent titled “tours” and the words are annotated to map back to their corresponding entities.
Appendix F

```javascript
function Tours(agent) {
    var date = agent.parameters.date;
    var time = agent.parameters.time;
    var number = agent.parameters.number;
    var Tour_Type = agent.parameters.Tour_Type;
    const taskKey = datastore.key('Tours');
    const entity = {
        key: taskKey,
        data: {
            item_name: 'Tours',
            date: date,
            time: time,
            order_time: new Date().toLocaleString(),
            number: number,
            Tour_Type: Tour_Type
        }
    };
    return datastore.save(entity).then(() => {
        console.log('Saved ${entity.key.name}: ${entity.data.item_name}');
        agent.add('Your tour has been scheduled on ${date} at ${time} for ${number} people.');
    });
}
```

This function is written in node.js script and to establish the entity variables which is written into the backend database. This function is adapted from code provided by Coursera in their course titled *Building Conversational Experiences with Dialogflow*
Appendix G

<table>
<thead>
<tr>
<th>Name/ID</th>
<th>Tour_Type</th>
<th>date</th>
<th>item_name</th>
<th>number</th>
<th>order_time</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>id=5632499082330112</td>
<td>group</td>
<td>2020-03-03T12:00:00-05:00</td>
<td>Tours</td>
<td>4</td>
<td>2020-03-02T14:00:00-05:00</td>
<td>2020-03-02T14:00:00-05:00</td>
</tr>
<tr>
<td>id=5634161670881280</td>
<td>individual</td>
<td>2020-02-13T12:00:00-05:00</td>
<td>Tours</td>
<td>2</td>
<td>2020-02-12T23:17:53</td>
<td>2020-02-13T02:00:00-05:00</td>
</tr>
<tr>
<td>id=5644004762845184</td>
<td>group</td>
<td>2020-02-17T12:00:00-05:00</td>
<td>Tours</td>
<td>5</td>
<td>2020-02-17T16:05:13</td>
<td>2020-02-17T14:00:00-05:00</td>
</tr>
</tbody>
</table>

This shows an example of the information stored in Google Datastore when a student books a tour using the chatbot created in this thesis.
Individual class
SUNY New Paltz will separately recognize your graduating class. These classes will be listed individually by school, graduation month, and year in the 2020 commencement program booklet. There will also be a point in the script where each group of graduates and candidates will be formally recognized by the president during the ceremony. The bookstore will also be offering graduation year "charms" that can be worn on the graduate cap tassel.

Lose tassel
You will need to buy a replacement at the campus bookstore. Special commencement hours to be announced.

Campus bookstore
No. You must purchase the cap and gown before the day of the ceremony.

Degree
If you are unsure of the titling of your specific degree, check with Records and Registration at 845-257-3100, or contact your adviser.

Diploma
No. The official diploma is not presented at Commencement. Diplomas will be mailed to the candidate’s permanent address approximately six-to-eight weeks after Commencement from the Records and Registration office. A diploma cannot be issued until all obligations, such as library fees, parking tickets, etc., are cleared. Note: When you cross the stage, you will be given a diploma cover which contains your Certification of Participation.

This is an excerpt of the SUNY New Paltz 2020 Commencement FAQ document, the corpus which serves as the backend knowledge base for this thesis.
This is an example of how the topics and definitions (parsed from the SUNY New Paltz 2020 Commencement FAQ document) appear in Google Datastore. Note that the definitions are encoded, but will be decoded when Google Dialogflow pulls them.
These are the training phrases used in the corpus based chatbot created to answer questions regarding the SUNY New Paltz 2020 commencement ceremony. These training phrases are for the intent titled “Topic” and the words are annotated to map back to the entity titled “topic”.

<table>
<thead>
<tr>
<th>PARAMETER NAME</th>
<th>ENTITY</th>
<th>RESOLVED VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic</td>
<td>@Topic</td>
<td>test</td>
</tr>
<tr>
<td>lodging</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diploma</td>
<td></td>
<td></td>
</tr>
<tr>
<td>receive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pre-order</td>
<td></td>
<td></td>
</tr>
<tr>
<td>residence hall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dorm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>status change</td>
<td></td>
<td></td>
</tr>
<tr>
<td>general parking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>disabilities</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>