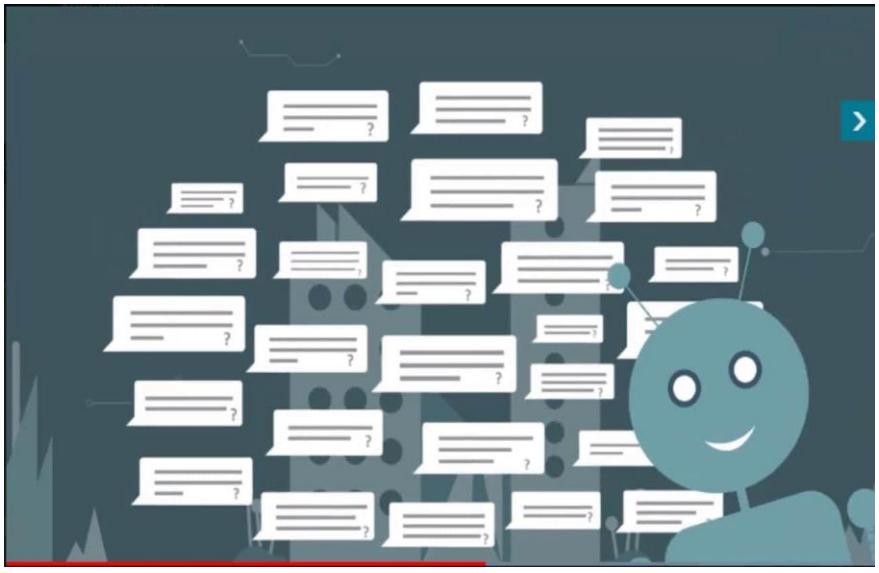
CHATBOT USING PYTHON PROJECT

Introduction:

In our increasingly digital world, chatbots have emerged as a revolutionary technology, transforming the way we interact with machines and access information. These computer programs, often powered by artificial intelligence, are designed to engage in text or voice-based conversations with users, simulating human-like interactions. Chatbots have rapidly gained prominence across a wide range of industries and applications, from streamlining customer support and enhancing e-commerce experiences to delivering healthcare information and offering educational assistance. In this discussion, we will delve into the world of chatbots, examining their various types, use cases, benefits, and challenges, to gain a comprehensive understanding of their impact on modern society.

Chatbots can be broadly categorized into rule-based systems, which operate within predefined guidelines and decision trees, and AI-powered chatbots, equipped with natural language processing and machine learning capabilities that enable them to comprehend and respond to user input in a more flexible and context-aware manner. Beyond text-based interactions, virtual assistants such as Siri, Google Assistant, and Alexa have expanded the chatbot concept to encompass a wide array of tasks and services, making them an integral part of our daily lives.

What is a chatbot?



At the most basic level, a chatbot is a computer program that simulates and processes human conversation (either written or spoken), allowing humans to interact with digital devices as if they were communicating with a real person. Chatbots can be as simple as rudimentary programs that answer a simple query with a single-line response, or as sophisticated as digital assistants that learn and evolve to deliver increasing levels of personalization as they gather and process information.

You've probably interacted with a chatbot whether you know it or not. For example, you're at your computer researching a product, and a window pops up on your screen asking if you need help. Or perhaps you're on your way to a concert and you use your smartphone to request a ride via chat. Or you might have used voice commands to order a coffee from your neighborhood café and received a response telling you when your order will be ready and what it will cost. These are all examples of scenarios in which you could be encountering a chatbot.

How do chatbots work?

Driven by AI, automated rules, natural-language processing (NLP), and machine learning (ML), chatbots process data to deliver responses to requests of all kinds.

There are two main types of chatbots.

- **Task-oriented (declarative) chatbots** are single-purpose programs that focus on performing one function. Using rules, NLP, and very little ML, they generate automated but conversational responses to user inquiries. Interactions with these chatbots are highly specific and structured and are most applicable to support and service functions—think robust, interactive FAQs. Task-oriented chatbots can handle common questions, such as queries about hours of business or simple transactions that don't involve a variety of variables. Though they do use NLP so end users can experience them in a conversational way, their capabilities are fairly basic. These are currently the most commonly used chatbots.

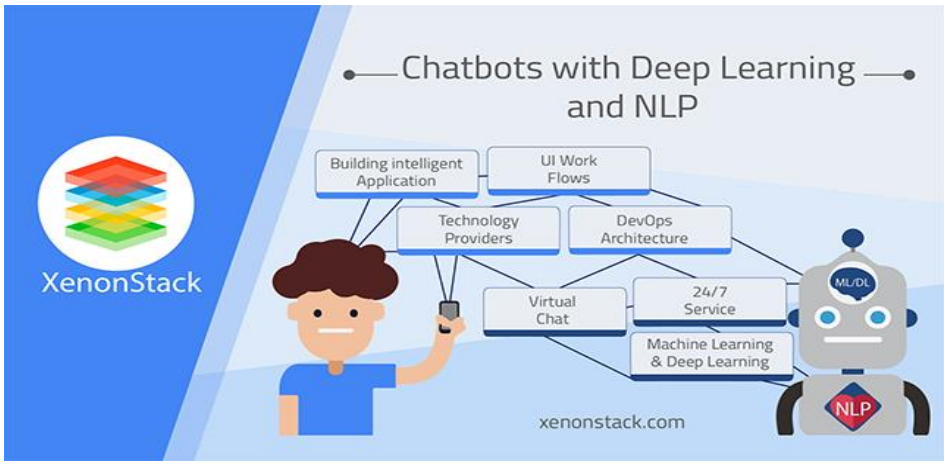
.

- **Data-driven and predictive (conversational) chatbots** are often referred to as virtual assistants or digital assistants, and they are much more sophisticated, interactive, and personalized than task-oriented chatbots. These chatbots are contextually aware and leverage natural-language understanding (NLU), NLP, and ML to learn as they go. They apply predictive intelligence and analytics to enable personalization based on user profiles and past user behavior. Digital assistants can learn a user's preferences over time, provide recommendations, and even anticipate needs. In addition to monitoring data and intent, they can initiate conversations. Apple's Siri and Amazon's Alexa are examples of consumer-oriented, data-driven, predictive chatbots.

Advanced digital assistants are also able to connect several single-purpose chatbots under one umbrella, pull disparate information from each of them, and then combine this information to perform a task while still maintaining context—so the chatbot doesn't become "confused." **(b) that runs on machine learning**

The following picture shows the working principle of chatbot using Artificial Intelligence.

Business Challenge for Building Chatbots



- Fixed set of answers
- Integration of ChatBot
- Understanding of Problem
- Security Issues
- Lack of Human Behaviour and Intentions
- Managing a ChatBot
- NLP limitations

Solution Offered for Implementing ChatBots with Deep Learning

Deep Learning which is galvanized by the functioning of the human brain, has composite engineering and used for the imitation of the data. Neural Network acts as the elementary brick of Deep Learning. A Neural Network is an Artificial Model of the human brain network modeled using hardware and software.

ChatBots Implementation Techniques

- Streaming of incoming data through the backend
- Create a model using Natural Language Processing
- Create a Natural Conversational Flow
- Add features to automate the process
- Invite Customers to Join

ChatBots Applications and Uses

ChatBot Applications are revolutionizing the world through efficient Human- Machine communication and Machine Learning Services. ChatBots covers an abundant audience including textual messages, Voice and Speech Conversations.

Advantages of ChatBot Applications involve -

- 24*7 Availability
- Platform Independent
- Time Saver
- Captures Onlookers
- Cost Efficient

Various ChatBot Applications -

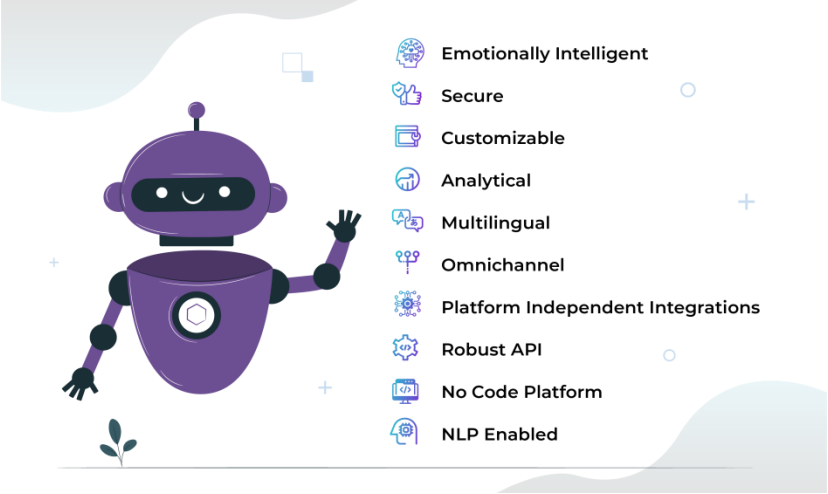
- Sales Lead Nurturing
- Client and Scrum Meetings
- HR Recruitment
- Website Enquiries
- Maintenance Support
- Emailing and Marketing
- Industries like HealthCare, Finance, Travel

ustomer experience. It’s a win/win proposition

Why were chatbots created?

Digitization is transforming society into a “mobile-first” population. As messaging applications grow in popularity, chatbots are increasingly playing an important role in this mobility-driven transformation. Intelligent conversational chatbots are often interfaces for mobile applications and are changing the way businesses and customers interact

Chatbots have become increasingly standard and valuable interfaces employed by numerous organizations for various purposes. They find numerous applications across different industries, such as providing personalized product recommendations to customers, offering round-the-clock customer support for query resolution, assisting with customer bookings, and much more. This article explores the process of creating a FAQ chatbot specifically designed for customer interaction. FAQ chatbots address questions within a specific domain, utilizing a predefined list of questions and corresponding answers. This type of chatbot relies on Semantic Question Matching as its underlying mechanism.



What Are The Types Of Chatbots?

Chatbots are classified primarily into two different types –

1. Rule-Based Chatbots

Rule-based chatbots are normally used for simpler applications and processes. These bots can answer questions based on a predefined set of rules that they have been programmed for and cannot operate on a standalone basis. These chatbots are only capable of executing rule-based commands and thus can lead to poor customer experiences.

2. AI Chatbots

AI chatbots are advanced and can handle open-ended queries wisely! They can become smarter over time using NLP and machine learning algorithms. AI-powered chatbot leverages natural language processing services model to learn the context and understand the meaning from the user input.

Why Does Your Business Need An AI-Powered Chatbot?

Still wondering why chatbots are important for your business. Then let's take a look at the top 10 reasons why chatbots can turn out to be a boon for your business–

1. Round the clock availability

Over 50% of customers expect a business to be open 24/7. (Oracle, 2017)

2. Faster resolution time

69% of consumers said they would prefer chatbots over human agents for receiving instantaneous responses. (Cognizant, 2019)

3. Cost-saving

Smart Chatbots can help businesses save costs up to 30% on their customer support costs. (IBM)

Chatbots allow businesses to connect with customers in a personal way without the expense of human representatives. For example, many of the questions or issues customers have are common and easily answered. That's why companies create FAQs and troubleshooting guides. Chatbots provide a personal alternative to a written FAQ or guide and can even triage questions, including handing off a customer issue to a live person if the issue becomes too complex for the chatbot to resolve. Chatbots have become popular as a time and money saver for businesses and an added convenience for customers.

How chatbots have evolved

The origin of the chatbot arguably lies with Alan Turing's 1950s vision of intelligent machines. Artificial intelligence, the foundation for chatbots, has progressed since that time to include superintelligent supercomputers such as IBM Watson.

The original chatbot was the phone tree, which led phone-in customers on an often cumbersome and frustrating path of selecting one option after another to wind their way through an automated customer service model. Enhancements in technology and the growing sophistication of AI, ML, and NLP evolved this model into pop-up, live, onscreen chats. And the evolutionary journey has continued.

With today's digital assistants, businesses can scale AI to provide much more convenient and effective interactions between companies and customers—directly from customers' digital devices.

Common chatbot uses

Chatbots are frequently used to improve the IT service management experience, which delves towards self-service and automating processes offered to internal staff. With an intelligent chatbot, common tasks such as password updates, system status, outage alerts, and knowledge management can be readily automated and made available 24/7, while broadening access to commonly used voice and text based conversational interfaces.

On the business side, chatbots are most commonly used in customer contact centers to manage incoming communications and direct customers to the appropriate resource. They're also frequently used for internal purposes, such as onboarding new employees and helping all employees with routine activities including vacation scheduling, training, ordering computers and business supplies, and other self-service activities that don't require human intervention.

On the consumer side, chatbots are performing a variety of customer services, ranging from ordering event tickets to booking and checking into hotels to comparing products and services. Chatbots are also commonly used to perform routine customer activities within the banking, retail, and food and beverage sectors. In addition, many public sector functions are enabled by chatbots, such as submitting requests for city services, handling utility-related inquiries, and resolving billing issues.

The future of chatbots

Where is the evolution of chatbots headed? Chatbots, like other AI tools, will be used to further enhance human capabilities and free humans to be more creative and innovative, spending more of their time on strategic rather than tactical activities.

In the near future, when AI is combined with the development of 5G technology, businesses, employees, and consumers are likely to enjoy enhanced chatbot features such as faster recommendations and predictions, and easy access to high-definition video conferencing from within a conversation. These and other possibilities are in the investigative stages and will evolve quickly as internet connectivity, AI, NLP, and ML advance. Eventually, every person can have a fully functional personal assistant right in their pocket, making our world a more efficient and connected place to live and work.

ChatterBot: Build a Chatbot With Python

Chatbots can provide real-time customer support and are therefore a valuable asset in many industries. When you understand the basics of the **ChatterBot** library, you can **build and train** a **self-learning chatbot** with just a few lines of Python code.

You'll get the basic chatbot up and running right away in step one, but the most interesting part is the learning phase, when you get to train your chatbot. The quality and preparation of your training data will make a big difference in your chatbot's performance.

To simulate a real-world process that you might go through to create an industry-relevant chatbot, you'll learn how to customize the chatbot's responses. You'll do this by preparing **WhatsApp chat data** to train the chatbot. You can apply a similar process to train your bot from different conversational data in any domain-specific topic.

In this tutorial, you'll learn how to:

- Build a **command-line chatbot** with ChatterBot
- **Train** the chatbot to customize its responses
- **Export** your WhatsApp chat history
- Perform **data cleaning** on the chat export using **regular expressions**
- **Retrain** the chatbot with industry-specific data

You'll also learn how ChatterBot stores your training data, and you'll find suggestions and pointers for next steps, so you can start collecting real user data and let the chatbot learn from it.

Overall, in this tutorial, you'll quickly run through the basics of creating a chatbot with ChatterBot and learn how Python allows you to get fun and useful results without needing to write a lot of code.

Source Code: Click here to download the free source code that you'll use to build a chatbot.

Demo

At the end of this tutorial, you'll have a command-line chatbot that can respond to your inputs with semi-meaningful replies:

You'll achieve that by preparing WhatsApp chat data and using it to train the chatbot. Beyond learning from your automated training, the chatbot will improve over time as it gets more exposure to questions and replies from user interactions.

Step 1: Create a Chatbot Using Python ChatterBot

In this step, you'll set up a virtual environment and install the necessary dependencies. You'll also create a working command-line chatbot that can reply to you—but it won't have very interesting replies for you yet.

To get started with your chatbot project, create and activate a virtual environment, then install `chatterbot` and `pytz`:

- Windows
- Linux + macOS

Windows PowerShell

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install chatterbot==1.0.4 pytz
```

Running these commands in your terminal application installs ChatterBot and its dependencies into a new Python virtual environment.

Python

```
# bot.py

from chatterbot import ChatBot

chatbot = ChatBot("Chatpot")

exit_conditions = (":q", "quit", "exit")
while True:
    query = input("> ")
```

```
10    if query in exit_conditions:
11        break
12    else:
13        print(f"🪴 {chatbot.get_response(query)}")
```

After importing `ChatBot` in line 3, you create an instance of `ChatBot` in line 5. The only required argument is a name, and you call this one `"Chatpot"`. No, that's not a typo—you'll actually build a chatty flowerpot chatbot in this tutorial! You'll soon notice that pots may not be the best conversation partners after all.

In line 8, you create a `while` loop that'll keep looping unless you enter one of the exit conditions defined in line 7. Finally, in line 13, you call `.get_response()` on the `ChatBot` instance that you created earlier and pass it the user input that you collected in line 9 and assigned to `query`.

The call to `.get_response()` in the final line of the short script is the only interaction with your `chatbot`. And yet—you have a functioning command-line chatbot that you can take for a spin.

When you run `bot.py`, ChatterBot might download some data and language models associated with the NLTK project. It'll print some information about that to your console. Python won't download this data again during subsequent runs.

Note: The NLTK project installs the data that ChatterBot uses into a default location on your operating system:

- **Windows:** `C:\nltk_data\`
- **Linux:** `/usr/share/nltk_data/`
- **macOS:** `/Users/<username>/nltk_data/`

NLTK will automatically create the directory during the first run of your chatbot.

If you're ready to communicate with your freshly homegrown `Chatpot`, then you can go ahead and run the Python file:

Shell

```
$ python bot.py
```

After the language models are set up, you'll see the greater than sign (`>`) that you defined in `bot.py` as your input prompt. You can now start to interact with your chatty pot:

Text

```
> hello
🪴 hello
> are you a plant?
🪴 hello
> can you chat, pot?
🪴 hello
```

Well … your chat-pot is *responding*, but it's really struggling to branch out. Tough to expect more from a potted plant—after all, it's never gotten to see the world!

Note: On Windows PowerShell, the potted plant emoji (🪴) might not render correctly. Feel free to replace it with any other prompt you like.

Even if your chat-pot doesn't have much to say yet, it's already learning and growing. To test this out, stop the current session. You can do this by typing one of the exit conditions—`":q"`, `"quit"`, or `"exit"`. Then start the chatbot another time. Enter a different message, and you'll notice that the chatbot remembers what you typed during the previous run:

```
Text
```

```
> hi
🪴 hello
> what's up?
🪴 are you a plant?
```

During the first run, ChatterBot created a SQLite database file where it stored all your inputs and connected them with possible responses. There should be three new files that have popped up in your working directory:

```
./
├── bot.py
├── db.sqlite3
├── db.sqlite3-shm
└── db.sqlite3-wal
```

ChatterBot uses the default `SQLStorageAdapter` and creates a SQLite file database unless you specify a different storage adapter.

Note: The main database file is `db.sqlite3`, while the other two, ending with `-wal` and `-shm`, are temporary support files.

Because you said both *hello* and *hi* at the beginning of the chat, your chat-pot learned that it can use these messages interchangeably. That means if you chat a lot with your new chatbot, it'll gradually have better replies for you. But improving its responses manually sounds like a long process!

Now that you've created a working command-line chatbot, you'll learn how to train it so you can have slightly more interesting conversations.

Step 2: Begin Training Your Chatbot

In the previous step, you built a chatbot that you could interact with from your command line. The chatbot started from a clean slate and wasn't very interesting to talk to.

In this step, you'll train your chatbot using `ListTrainer` to make it a little smarter from the start. You'll also learn about built-in trainers that come with ChatterBot, including their limitations.

Your chatbot doesn't have to start from scratch, and ChatterBot provides you with a quick way to train your bot. You'll use ChatterBot's `ListTrainer` to provide some conversation samples that'll give your chatbot more room to grow:

Python

```
# bot.py

from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer

chatbot = ChatBot("Chatpot")

trainer = ListTrainer(chatbot)
trainer.train([
    "Hi",
    "Welcome, friend ",
])
trainer.train([
    "Are you a plant?",
    "No, I'm the pot below the plant!",
])

exit_conditions = (":q", "quit", "exit")
while True:
    query = input("> ")
    if query in exit_conditions:
        break
    else:
        print(f" {chatbot.get_response(query)}")
```

In line 4, you import `ListTrainer`, to which you pass your `chatbot` on line 8 to create `trainer`.

In lines 9 to 12, you set up the first training round, where you pass a list of two strings to `trainer.train()`. Using `.train()` injects entries into your database to build upon the graph structure that ChatterBot uses to choose possible replies.

Note: If you pass an iterable with exactly two items to `ListTrainer.train()`, then ChatterBot considers the first item a statement and the second item an acceptable response.

You can run more than one training session, so in lines 13 to 16, you add another statement and another reply to your chatbot's database.

If you now run the interactive chatbot once again using `python bot.py`, you can elicit somewhat different responses from it than before:

```
Text
```

```
> hi
 Welcome, friend 
> hello
 are you a plant?
> me?
 are you a plant?
> yes
 hi
> are you a plant?
 No, I'm the pot below the plant!
> cool
 Welcome, friend 
```

The conversation isn't yet fluent enough that you'd like to go on a second date, but there's additional context that you didn't have before! When you train your chatbot with more data, it'll get better at responding to user inputs.

The ChatterBot library comes with some corpora that you can use to train your chatbot. However, at the time of writing, there are some issues if you try to use these resources straight out of the box.

While the provided corpora might be enough for you, in this tutorial you'll skip them entirely and instead learn how to adapt your own conversational input data for training with ChatterBot's `ListTrainer`.

To train your chatbot to respond to industry-relevant questions, you'll probably need to work with custom data, for example from existing support requests or chat logs from your company.

Moving forward, you'll work through the steps of converting chat data from a WhatsApp conversation into a format that you can use to train your chatbot. If your own resource is WhatsApp conversation data, then you can use these steps directly. If your data comes from elsewhere, then you can adapt the steps to fit your specific text format.

To start off, you'll learn how to export data from a WhatsApp chat conversation.

Step 3: Export a WhatsApp Chat

At the end of this step, you'll have downloaded a TXT file that contains the chat history of a WhatsApp conversation. If you don't have a WhatsApp account or don't want to work with your own conversational data, then you can download a sample chat export below:

Source Code: Click here to download the free source code that you'll use to build a chatbot.

If you're going to work with the provided chat history sample, you can skip to the next section, where you'll clean your chat export.

To export the history of a conversation that you've had on WhatsApp, you need to open the conversation on your phone. Once you're on the conversation screen, you can access the export menu:

1. Click on the three dots (⋮) in the top right corner to open the main menu.
2. Choose *More* to bring up additional menu options.
3. Select *Export chat* to create a TXT export of your conversation.

In the stitched-together screenshots below, you can see the three consecutive steps numbered and outlined in red:

Once you've clicked on *Export chat*, you need to decide whether or not to include media, such as photos or audio messages. Because your chatbot is only dealing with text, select *WITHOUT MEDIA*. Then, you can declare where you'd like to send the file.

Again, you can see an example of these next steps in two stitched-together WhatsApp screenshots with red numbers and outlines below:

In this example, you saved the chat export file to a Google Drive folder named *Chat exports*. You'll have to set up that folder in your Google Drive before you can select it as an option. Of course, you don't need to use Google Drive. As long as you save or send your chat export file so that you can access to it on your computer, you're good to go.

Once that's done, switch back to your computer. Find the file that you saved, and download it to your machine.

Specifically, you should save the file to the folder that also contains `bot.py` and rename it `chat.txt`. Then, open it with your favorite text editor to inspect the data that you received:

```
Text
```

```
9/15/22, 14:50 - Messages and calls are end-to-end encrypted.
↪No one outside of this chat, not even WhatsApp, can read
↪or listen to them. Tap to learn more.
```

```
9/15/22, 14:49 - Philipp: Hi Martin, Philipp here!
9/15/22, 14:50 - Philipp: I'm ready to talk about plants!
9/15/22, 14:51 - Martin: Oh that's great!
9/15/22, 14:52 - Martin: I've been waiting for a good convo about
↳plants for a long time
9/15/22, 14:52 - Philipp: We all have.
9/15/22, 14:52 - Martin: Did you know they need water to grow?
...
```

If you remember how ChatterBot handles training data, then you'll see that the format isn't ideal to use for training.

ChatterBot uses complete lines as messages when a chatbot replies to a user message. In the case of this chat export, it would therefore include all the message metadata. That means your friendly pot would be studying the dates, times, and usernames! Not exactly great conversation fertilizer.

To avoid this problem, you'll clean the chat export data before using it to train your chatbot.

Step 4: Clean Your Chat Export

In this step, you'll clean the WhatsApp chat export data so that you can use it as input to train your chatbot on an industry-specific topic. In this example, the topic will be … houseplants!

Most data that you'll use to train your chatbot will require some kind of cleaning before it can produce useful results. It's just like the old saying goes:

Garbage in, garbage out (Source)

Take some time to explore the data that you're working with and to identify potential issues:

Text

```
9/15/22, 14:50 - Messages and calls are end-to-end encrypted.
↳No one outside of this chat, not even WhatsApp, can read
↳or listen to them. Tap to learn more.

...

9/15/22, 14:50 - Philipp: I'm ready to talk about plants!

...
```

```
9/16/22, 06:34 - Martin: <Media omitted>


...
```

For example, you may notice that the first line of the provided chat export isn't part of the conversation. Also, each actual message starts with metadata that includes a date, a time, and the username of the message sender.

If you scroll further down the conversation file, you'll find lines that aren't real messages. Because you didn't include media files in the chat export, WhatsApp replaced these files with the text `<Media omitted>`.

All of this data would interfere with the output of your chatbot and would certainly make it sound much less conversational. Therefore, it's a good idea to remove this data.

Open up a new Python file to preprocess your data before handing it to ChatterBot for training. Start by reading in the file content and removing the chat metadata:

Python

```python
# cleaner.py

import re

def remove_chat_metadata(chat_export_file):
    date_time = r"(\d+\/\d+\/\d+,\s\d+:\d+)"  # e.g. "9/16/22, 06:34"
    dash_whitespace = r"\s-\s"  # " - "
    username = r"([\w\s]+)"  # e.g. "Martin"
    metadata_end = r":\s"  # ": "
    pattern = date_time + dash_whitespace + username + metadata_end

    with open(chat_export_file, "r") as corpus_file:
        content = corpus_file.read()
    cleaned_corpus = re.sub(pattern, "", content)
    return tuple(cleaned_corpus.split("\n"))

if __name__ == "__main__":
    print(remove_chat_metadata("chat.txt"))
```

This function removes conversation-irrelevant message metadata from the chat export file using the built-in `re` module, which allows you to work with regular expressions:

- **Line 3** imports `re`.
- **Lines 6 to 9** define multiple regex patterns. Constructing multiple patterns helps you keep track of what you're matching and gives you the flexibility to use the separate capturing groups to apply further preprocessing later on. For example, with access to `username`, you could chunk conversations by merging messages sent consecutively by the same user.
- **Line 10** concatenates the regex patterns that you defined in lines 6 to 9 into a single pattern. The complete pattern matches all the metadata that you want to remove.
- **Lines 12 and 13** open the chat export file and read the data into memory.
- **Line 14** uses `re.sub()` to replace each occurrence of the pattern that you defined in `pattern` with an empty string (`""`), effectively deleting it from the string.
- **Line 15** first splits the file content string into list items using `.split("\n")`. This breaks up `cleaned_corpus` into a list where each line represents a separate item. Then, you convert this list into a tuple and return it from `remove_chat_metadata()`.
- **Lines 17 and 18** use Python's name-main idiom to call `remove_chat_metadata()` with `"chat.txt"` as its argument, so that you can inspect the output when you run the script.

Eventually, you'll use `cleaner` as a module and import the functionality directly into `bot.py`. But while you're developing the script, it's helpful to inspect intermediate outputs, for example with a `print()` call, as shown in line 18.

Note: It's a good idea to run your script often while you're developing the code. As an alternative to printing the output, you could use `breakpoint()` to inspect your code with `pdb`. If you use a debugger such as `pdb`, then you can interact with the code objects rather than just printing a static representation.

After removing the message metadata from each line, you also want to remove a few complete lines that aren't relevant for the conversation. To do this, create a second function in your data cleaning script:

Python

```
1# cleaner.py
2
3# ...
4
5def remove_non_message_text(export_text_lines):
6    messages = export_text_lines[1:-1]
7
8    filter_out_msgs = ("<Media omitted>",)
9    return tuple((msg for msg in messages if msg not in filter_out_msgs))
10
11if __name__ == "__main__":
12    message_corpus = remove_chat_metadata("chat.txt")
13    cleaned_corpus = remove_non_message_text(message_corpus)
14    print(cleaned_corpus)
```

In `remove_non_message_text()`, you've written code that allows you to remove irrelevant lines from the conversation corpus:

- **Line 6** removes the first introduction line, which every WhatsApp chat export comes with, as well as the empty line at the end of the file.
- **Line 8** creates a tuple where you can define what strings you want to exclude from the data that'll make it to training. For now, it only contains one string, but if you wanted to remove other content as well, you could quickly add more strings to this tuple as items.
- **Line 9** filters `messages` for the strings defined in `filter_out_msgs` using a generator expression that you convert to a tuple before returning it.

Finally, you've also changed lines 12 to 14. You now collect the return value of the first function call in the variable `message_corpus`, then use it as an argument to `remove_non_message_text()`. You save the result of that function call to `cleaned_corpus` and print that value to your console on line 14.

Because you want to treat `cleaner` as a module and run the cleaning code in `bot.py`, it's best to now refactor the code in the name-main idiom into a main function that you can then import and call in `bot.py`:

Python

```
1# cleaner.py
2
3import re
4
5def clean_corpus(chat_export_file):
6    message_corpus = remove_chat_metadata(chat_export_file)
7    cleaned_corpus = remove_non_message_text(message_corpus)
8    return cleaned_corpus
9
10# ...
11
12# Deleted: if __name__ == "__main__":
```

You refactor your code by moving the function calls from the name-main idiom into a dedicated function, `clean_corpus()`, that you define toward the top of the file. In line 6, you replace `"chat.txt"` with the parameter `chat_export_file` to make it more general. You'll provide the filename when calling the function. The `clean_corpus()` function returns the cleaned corpus, which you can use to train your chatbot.

After creating your cleaning module, you can now head back over to `bot.py` and integrate the code into your pipeline.

Step 5: Train Your Chatbot on Custom Data and Start Chatting

In this step, you'll train your chatbot with the WhatsApp conversation data that you cleaned in the previous step. You'll end up with a chatbot that you've trained on industry-specific conversational data, and you'll be able to chat with the bot—about houseplants!

Open up `bot.py` and include calls to your cleaning functions in the code:

Python

```
# bot.py

from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
from cleaner import clean_corpus

CORPUS_FILE = "chat.txt"

chatbot = ChatBot("Chatpot")

trainer = ListTrainer(chatbot)
cleaned_corpus = clean_corpus(CORPUS_FILE)
trainer.train(cleaned_corpus)

exit_conditions = (":q", "quit", "exit")
while True:
    query = input("> ")
    if query in exit_conditions:
        break
    else:
        print(f"🪴 {chatbot.get_response(query)}")
```

You don't need to do a lot of extra work in this file, because you've prepared your cleaning functions well:

- **Line 5** imports `clean_corpus()` from `cleaner`.
- **Line 7** defines the filename of your chat export file. In this example, you assume that it's called `"chat.txt"`, and it's located in the same directory as `bot.py`. If you need more advanced path handling, then take a look at Python's `pathlib` module.
- **Line 12** applies your cleaning code to the chat history file and returns a tuple of cleaned messages, which you call `cleaned_corpus`.

- **Line 13** finally uses that data as input to `.train()`, effectively training your chatbot with the WhatsApp conversation data.

```
Text
```

```
> hi
 Welcome, friend 
> thanks for the green welcome
 I let you
> you let me be here?
 It's a monsters!
> did you mean monstera?
 The leafs are getting dryer and dryer. But she's growing new ones
> who?
 Do raindrops touch their leaves?
> very philosophical!
 Lol
> ;)
 I don't grow any crop at home
> no crops in pots
 Ah, gotcha!
> ah, I just thought you meant that
 But water makes sense
> yes, water in pots
 Do you have any plant care pro tips?
> water.
 I always thought that love and music was more than enough
> haha
 Well, no. Just the balcony. This worked best
> ok, balcony is notes
 I let you
```

How to Build Your AI Chatbot with NLP in Python?

In this article, we will create an AI chatbot using Natural Language Processing (NLP) in Python. Our goal is to help you build a smart chatbot. First, we'll explain NLP, which helps computers understand human language. Then, we'll show you how to use AI to make a chatbot to have real conversations with people. Finally, we'll talk about the tools you need to create a chatbot like ALEXA or Siri.

This guide is hands-on. We'll give you step-by-step instructions, and you can follow our examples or change them to fit your needs. So, let's begin this journey into the world of NLP and AI chatbots!

This article was published as a part of the Data Science Blogathon.

Were you ever curious as to how to build a talking ChatBot with Python and also have a conversation with your own personal AI?

As the topic suggests we are here to help you have a conversation with your AI today. To have a conversation with your AI, you need a few pre-trained tools which can help you build an AI chatbot system. In this article, we will guide you to combine speech recognition processes with an artificial intelligence algorithm.

Natural Language Processing or NLP is a prerequisite for our project. NLP allows computers and algorithms to understand human interactions via various languages. In order to process a large amount of natural language data, an AI will definitely need NLP or Natural Language Processing. Currently, we have a number of NLP research ongoing in order to improve the AI chatbots and help them understand the complicated nuances and undertones of human conversations.

Chatbots are applications that businesses and other entities employ to facilitate automated conversations between AI and humans. These conversations can occur through text or speech. Chatbots must

comprehend and imitate human conversation when engaging with people worldwide. Chatbots have made significant progress from ELIZA, the first-ever chatbot, to today's Amazon ALEXA. This tutorial will cover all the fundamental concepts necessary for creating a basic chatbot that can comprehend and respond to human interactions. We will utilize speech recognition APIs and pre-trained Transformer models.

What is NLP?

NLP, or Natural Language Processing, stands for teaching machines to understand human speech and spoken words. NLP combines computational linguistics, which involves rule-based modeling of human language, with intelligent algorithms like statistical, machine, and deep learning algorithms. Together, these technologies create the smart voice assistants and chatbots we use daily.

In human speech, there are various errors, differences, and unique intonations. NLP technology empowers machines to rapidly understand, process, and respond to large volumes of text in real-time. You've likely encountered NLP in voice-guided GPS apps, virtual assistants, speech-to-text note creation apps, and other chatbots that offer app support in your everyday life. In the business world, NLP is instrumental in streamlining processes, monitoring employee productivity, and enhancing sales and after-sales efficiency.

Tasks in NLP

Interpreting and responding to human speech presents numerous challenges, as discussed in this article. Humans take years to conquer these challenges when learning a new language from scratch. Programmers have integrated various functions into NLP technology to tackle these hurdles and create practical tools for understanding human speech, processing it, and generating suitable responses.

NLP tasks involve breaking down human text and audio signals from voice data in ways that computers can analyze and convert into comprehensible data. Some of the tasks in NLP data ingestion include:

1. **Speech Recognition:** This process involves converting speech into text, a crucial step in speech analysis. Within speech recognition, there is a subprocess called speech tagging, which allows a computer to break down speech and add context, accents, or other speech attributes.
2. **Word Sense Disambiguation:** In human speech, a word can have multiple meanings. Word sense disambiguation is a semantic analysis that selects the most appropriate meaning for a word based on its context. For instance, it helps determine whether a word functions as a verb or a pronoun.

3. **Named Entity Recognition (NER):** NER identifies words and phrases as specific entities, such as recognizing “Dev” as a person’s name or “America” as the name of a country.
4. **Sentiment Analysis:** Human speech often contains sentiments and undertones. Extracting these nuances and hidden emotions, like attitude, sarcasm, fear, or joy, is one of the most challenging tasks undertaken by NLP processes.

Types of AI Chatbots

Chatbots are a relatively recent concept and despite having a huge number of programs and NLP tools, we basically have just two different categories of chatbots based on the NLP technology that they utilize. These two types of chatbots are as follows:

Scripted Chatbots

Scripted chatbots are chatbots that operate based on pre-determined scripts stored in their library. When a user inputs a query, or in the case of chatbots with speech-to-text conversion modules, speaks a query, the chatbot replies according to the predefined script within its library. One drawback of this type of chatbot is that users must structure their queries very precisely, using comma-separated commands or other regular expressions, to facilitate string analysis and understanding. This makes it challenging to integrate these chatbots with NLP-supported speech-to-text conversion modules, and they are rarely suitable for conversion into intelligent virtual assistants.

Artificially Intelligent Chatbots

Artificially intelligent chatbots, as the name suggests, are designed to mimic human-like traits and responses. NLP (Natural Language Processing) plays a significant role in enabling these chatbots to understand the nuances and subtleties of human conversation. When NLP is combined with artificial intelligence, it results in truly intelligent chatbots capable of responding to nuanced questions and learning from each interaction to provide improved responses in the future. AI chatbots find applications in various platforms, including automated chat support and virtual assistants designed to assist with tasks like recommending songs or restaurants.

Installing Packages required to Build AI Chatbot

We will begin by installing a few libraries which are as follows :

Code:

```
# To be able to convert text to Speech
! pip install SpeechRecognition  #(3.8.1)
#To convey the Speech to text and also speak it out
!pip install gTTS  #(2.2.3)
# To install our language model
!pip install transformers  #(4.11.3)
!pip install tensorflow #(2.6.0, or pytorch)
```

We will start by importing some basic functions:

```
import numpy as np
```

We will begin by creating an empty class which we will build step by step. To build the chatbot, we would need to execute the full script. The name of the bot will be " Dev"

```
# Beginning of the AI
class ChatBot():
    def __init__(self, name):
        print("----- starting up", name, "-----")
        self.name = name
# Execute the AI
if __name__ == "__main__":
    ai = ChatBot(name="Dev")
```

What is Speech Recognition?

NLP or Natural Language Processing has a number of subfields as conversation and speech are tough for computers to interpret and respond to. One such subfield of NLP is Speech Recognition. Speech Recognition works with methods and technologies to enable recognition and translation of human spoken languages into something that the computer or AI can understand and respond to.

For computers, understanding numbers is easier than understanding words and speech. When the first few speech recognition systems were being created, IBM Shoebox was the first to get decent success with understanding and responding to a select few English words. Today, we have a number of successful examples which understand myriad languages and respond in the correct dialect and language as the human interacting with it. Most of this success is through the SpeechRecognition library.

Using Google APIs

To use popular Google APIs we will use the following code:

Code:

```
import speech_recognition as sr
def speech_to_text(self):
```

```
    recognizer = sr.Recognizer()
    with sr.Microphone() as mic:
         print("listening...")
         audio = recognizer.listen(mic)
    try:
         self.text = recognizer.recognize_google(audio)
         print("me --> ", self.text)
    except:
         print("me -->  ERROR")
```

Note: The first task that our chatbot must work for is the speech to text conversion. Basically, this involves converting the voice or audio signals into text data. In summary, the chatbot actually 'listens' to your speech and compiles a text file containing everything it could decipher from your speech. You can test the codes by running them and trying to say something aloud. It should optimally capture your audio signals and convert them into text.

Speech to Text Conversion

```
# Execute the AI
if __name__ == "__main__":
     ai = ChatBot(name="Dev")
     while True:
         ai.speech_to_text()
```

Output :

```
----- starting up Dev -----
listening...
me --> Try to say something
```

Note: Here I am speaking and not typing

Processing Suitable Responses

Next, our AI needs to be able to respond to the audio signals that you gave to it. In simpler words, our chatbot has received the input. Now, it must process it and come up with suitable responses and be able to give output or response to the human speech interaction. To follow along, please add the following function as shown below. This method ensures that the chatbot will be activated by speaking its name. When you say "Hey Dev" or "Hello Dev" the bot will become active.

Code:

```
def wake_up(self, text):
    return True if self.name in text.lower() else False
```

As a cue, we give the chatbot the ability to recognize its name and use that as a marker to capture the following speech and respond to it accordingly. This is done to make sure that the chatbot doesn't

respond to everything that the humans are saying within its 'hearing' range. In simpler words, you wouldn't want your chatbot to always listen in and partake in every single conversation. Hence, we create a function that allows the chatbot to recognize its name and respond to any speech that follows after its name is called.

Fine-tuning Bot Responses

After the chatbot hears its name, it will formulate a response accordingly and say something back. For this, the chatbot requires a text-to-speech module as well. Here, we will be using GTTS or Google Text to Speech library to save mp3 files on the file system which can be easily played back.

The following functionality needs to be added to our class so that the bot can respond back

Code:

```
from gtts import gTTS
import os
@staticmethod
def text_to_speech(text):
    print("AI --> ", text)
    speaker = gTTS(text=text, lang="en", slow=False)
    speaker.save("res.mp3")
    os.system("start res.mp3")  #if you have a macbook->afplay or for windows use-
>start
    os.remove("res.mp3")
```

Code :

```
#Those two functions can be used like this
# Execute the AI
if __name__ == "__main__":
     ai = ChatBot(name="Dev")
     while True:
         ai.speech_to_text()
         ## wake up
         if ai.wake_up(ai.text) is True:
             res = "Hello I am Dev the AI, what can I do for you?"
         ai.text_to_speech(res)
```

Output :

```
----- starting up Dev -----
listening...
me --> Hey Dev
AI --> Hello I am Dev the AI, what can I do for you?
```

Next, we can consider upgrading our chatbot to do simple commands like some o the virtual assistants help you to do. An example of such a task would be to equip the chatbot to be able to answer correctly whenever the user asks for the current time. To add this function to the chatbot class, follow along with the code given below:

Code:

```
import datetime
@staticmethod
def action_time():
    return datetime.datetime.now().time().strftime('%H:%M')
#and run the script after adding the above function to the AI class
# Run the AI
if __name__ == "__main__":
ai = ChatBot(name="Dev")
while True:
        ai.speech_to_text()
        ## waking up
        if ai.wake_up(ai.text) is True:
            res = "Hello I am Dev the AI, what can I do for you?"
        ## do any action
        elif "time" in ai.text:
           res = ai.action_time()
        ## respond politely
        elif any(i in ai.text for i in ["thank","thanks"]):
           res = np.random.choice(
                 ["you're welcome!","anytime!",
                  "no problem!","cool!",
                  "I'm here if you need me!","peace out!"])
        ai.text_to_speech(res)
```

Output :

```
----- starting up Dev -----
listening...
me --> Hey Dev
AI --> Hello I am Dev the AI, what can I do for you?
listening...
me --> What is the time
AI --> 17:30
listening...
me --> thanks
AI --> cool!
listening...
```

After all of the functions that we have added to our chatbot, it can now use speech recognition techniques to respond to speech cues and reply with predetermined responses. However, our chatbot is still not very intelligent in terms of responding to anything that is not predetermined or preset. It is now time to incorporate artificial intelligence into our chatbot to create intelligent responses to human speech interactions with the chatbot or the ML model trained using NLP or Natural Language Processing.

The Language Model for AI Chatbot

Here, we will use a Transformer Language Model for our chatbot. This model was presented by Google and it replaced the earlier traditional sequence to sequence models with attention mechanisms. This language model dynamically understands speech and its undertones. Hence, the model easily performs NLP tasks. Some of the most popularly used language models are Google's BERT and OpenAI's GPT. These models have multidisciplinary functionalities and billions of parameters which helps to improve the.

Image 5

This is where the chatbot becomes intelligent and not just a scripted bot that will be ready to handle any test thrown at them. The main package that we will be using in our code here is the Transformers package provided by HuggingFace. This tool is popular amongst developers as it provides tools that are pre-trained and ready to work with a variety of NLP tasks. In the code below, we have specifically used the DialogGPT trained and created by Microsoft based on millions of conversations and ongoing chats on the Reddit platform in a given interval of time.

Code:

```
import transformers
```

```
nlp = transformers.pipeline("conversational",
                            model="microsoft/DialoGPT-medium")
#Time to try it out
input_text = "hello!"
nlp(transformers.Conversation(input_text), pad_token_id=50256)
```

Reminder: Don't forget to provide the pad_token_id as the current version of the library we are using in our code raises a warning when this is not specified. What you can do to avoid this warning is to add this as a parameter.

nlp(transformers.Conversation(input_text), pad_token_id=50256)

You will get a whole conversation as the pipeline output and hence you need to extract only the response of the chatbot here.

Code:

```
chat = nlp(transformers.Conversation(ai.text), pad_token_id=50256)
res = str(chat)
res = res[res.find("bot >> ")+6:].strip()
```

Finally, we're ready to run the Chatbot and have a fun conversation with our AI. Here's the full code:

```
----- starting up Dev -----
listening...
me --> Hello!
AI --> Hello :D
listening...
```

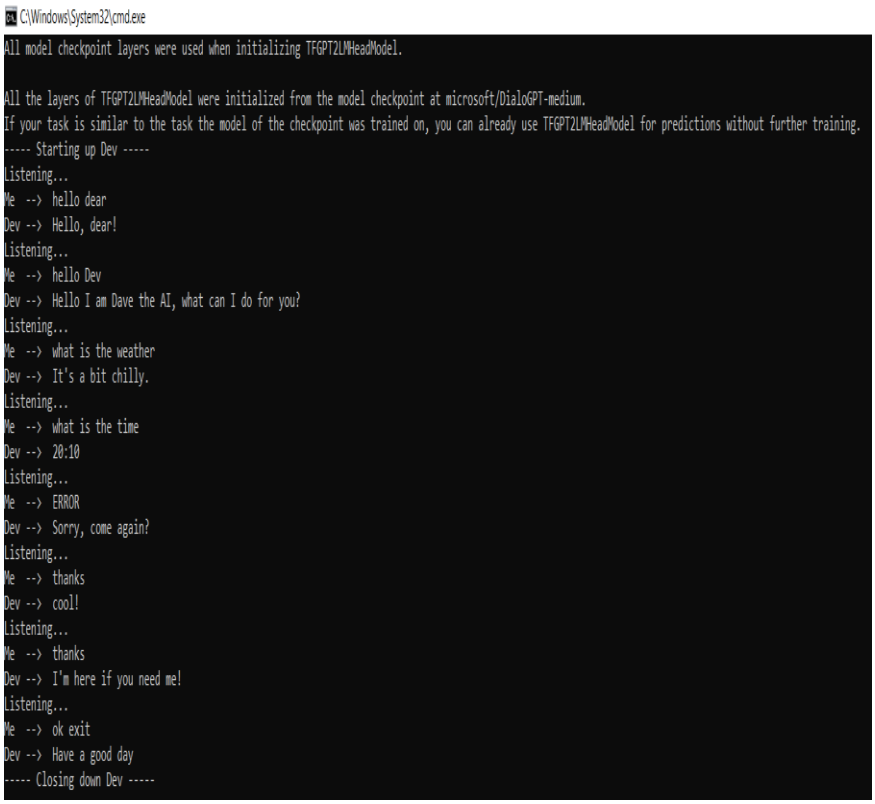
Great! The bot can both perform some specific tasks like a virtual assistant (i.e. saying the time when asked) and have casual conversations. And if you think that Artificial Intelligence is here to stay, she agrees:

Final Code:

```
# for speech-to-text
import speech_recognition as sr
# for text-to-speech
from gtts import gTTS
# for language model
import transformers
import os
import time
# for data
import os
import datetime
import numpy as np
# Building the AI
```

```
class ChatBot():
    def __init__(self, name):
        print("----- Starting up", name, "-----")
        self.name = name
    def speech_to_text(self):
        recognizer = sr.Recognizer()
        with sr.Microphone() as mic:
            print("Listening...")
            audio = recognizer.listen(mic)
            self.text="ERROR"
        try:
            self.text = recognizer.recognize_google(audio)
            print("Me  --> ", self.text)
        except:
            print("Me  -->  ERROR")
    @staticmethod
    def text_to_speech(text):
        print("Dev --> ", text)
        speaker = gTTS(text=text, lang="en", slow=False)
        speaker.save("res.mp3")
        statbuf = os.stat("res.mp3")
        mbytes = statbuf.st_size / 1024
        duration = mbytes / 200
        os.system('start res.mp3')  #if you are using mac->afplay or else for
windows->start
        # os.system("close res.mp3")
        time.sleep(int(50*duration))
        os.remove("res.mp3")
    def wake_up(self, text):
        return True if self.name in text.lower() else False
    @staticmethod
    def action_time():
        return datetime.datetime.now().time().strftime('%H:%M')
# Running the AI
if __name__ == "__main__":
    ai = ChatBot(name="dev")
    nlp = transformers.pipeline("conversational", model="microsoft/DialoGPT-medium")
    os.environ["TOKENIZERS_PARALLELISM"] = "true"
    ex=True
    while ex:
        ai.speech_to_text()
        ## wake up
        if ai.wake_up(ai.text) is True:
            res = "Hello I am Dave the AI, what can I do for you?"
        ## action time
        elif "time" in ai.text:
            res = ai.action_time()
        ## respond politely
        elif any(i in ai.text for i in ["thank","thanks"]):
            res = np.random.choice(["you're welcome!","anytime!","no
problem!","cool!","I'm here if you need me!","mention not"])
        elif any(i in ai.text for i in ["exit","close"]):
            res = np.random.choice(["Tata","Have a good day","Bye","Goodbye","Hope to
meet soon","peace out!"])
            ex=False
```

```
        ## conversation
        else:
            if ai.text=="ERROR":
                res="Sorry, come again?"
            else:
                chat = nlp(transformers.Conversation(ai.text), pad_token_id=50256)
                res = str(chat)
                res = res[res.find("bot >> ")+6:].strip()
        ai.text_to_speech(res)
    print("----- Closing down Dev -----")
```



To run a file and install the module, use the command “python3.9” and “pip3.9” respectively if you have more than one version of python for development purposes. “PyAudio” is another troublesome module and you need to manually google and find the correct “.whl” file for your version of Python and install it using pip.

A. To create an NLP chatbot, define its scope and capabilities, collect and preprocess a dataset, train an NLP model, integrate it with a messaging platform, develop a user interface, and test and refine the chatbot based on feedback. Tools such as Dialogflow, IBM Watson Assistant, and Microsoft Bot Framework offer pre-built models and integrations to facilitate development and deployment.

The media shown in this article on AI Chatbot is not owned by Analytics Vidhya and are used at the Author's discretion.

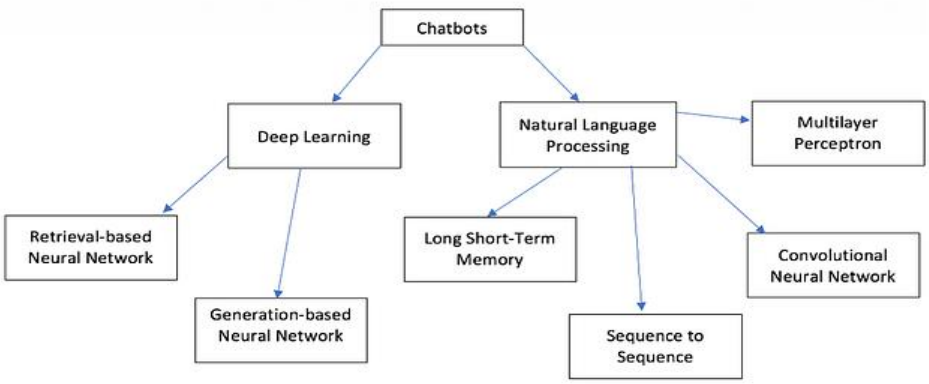
Related

Conversational AI Chatbot using Deep Learning: How Bi-directional LSTM, Machine Reading Comprehension, Transfer Learning, Sequence to Sequence Model with multi-headed attention mechanism, Generative Adversarial Network, Self Learning based Sentiment Analysis and Deep Reinforcement Learning can help in Dialog Management for Conversational AI chatbot

keywords: *NLU, NLG, Word Embedding, Tensorflow, RNN, Bi-directional LSTM, Generative Adversarial Network, Machine Reading Comprehension, Transfer Learning, Sequence to Sequence Model with multi-headed attention mechanism, Deep*

Reinforcement Learning, Self-learning based on Sentiment Analysis, Knowledge base, Recurrent Embedding Dialogue policy, Dual Encoder LSTM, Encoder-Decoder

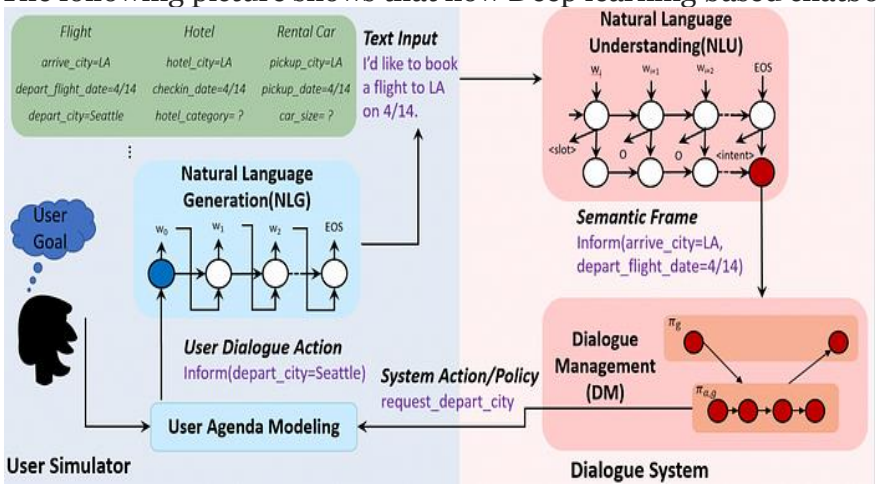
The below picture illustrate the conceptual map of Chatbot using Deep learning,



source

The chatbot needs to be able to understand the intentions of the sender's message, determine what type of response message (a follow-up question, direct response, etc.) is required, and follow correct grammatical and lexical rules while forming the response. Some models may use additional meta information from data, such as speaker id, gender, emotion. Sometimes, sentiment analysis is used to allows the chatbot to 'understand' the mood of the user by analysing verbal and sentence structuring clues.

The following picture shows that how Deep learning based chatbot work internally,



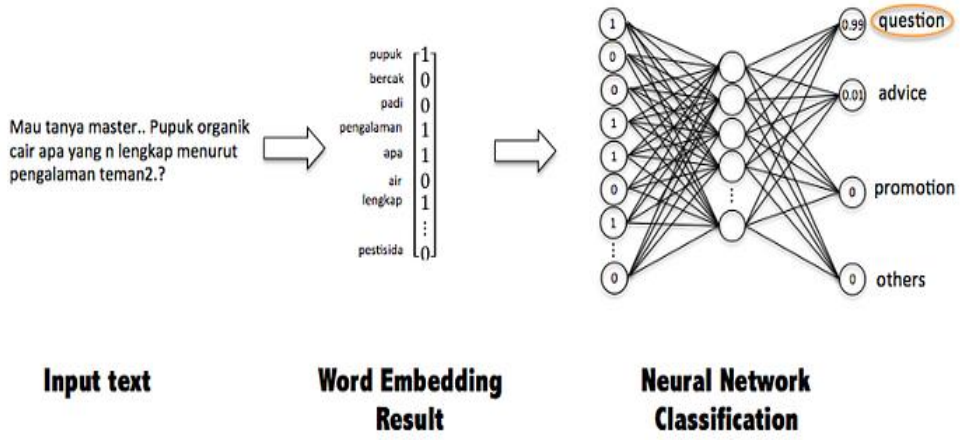
source

2. Role of NLU, NLG and Dialogue Management in Conversational AI

Natural Language Understanding

The NLU unit is responsible for transforming the user utterance to a predefined semantic frame according to the system's conventions, i.e. to a format understandable for the system. This includes a task of slot filling and intent detection. For example, the intent, could be a greeting, like Hello, Hi, Hey, or it could have an inform nature, for example I like Indian food, where the user is giving some additional information. Depending on the interests, the slots could be very diverse, like the actor name, price, start time, destination city etc. As we can see, the intents and the slots are defining the closed-domain nature of the Chatbot. The task of slot filling and intent detection is seen as a sequence tagging problem. For this reason, the NLU component is usually implemented as an LSTM-based recurrent neural network with a Conditional Random Field (CRF) layer on top of it. The model presented is a sequence-to-sequence model using bidirectional LSTM network, which fills the slots and predicts the intent in the same time. On the other hand, the model is doing the same using an attention-based RNN. To achieve such a task, the dataset labels consist of: concatenated B–I–O (Begin, Inside, Outside) slot tags, the intent tag and an additional end-of-string (EOS) tag. As an example, in a restaurant reservation scenario, given the sentence Are there any French restaurants in Toronto downtown?, the task is to correctly output, or fill, the following slots: {cuisine: French} and {location: Toronto downtown}.

The following picture shows the classification process for intent classification using Neural Network as,



source

Natural Language Generator (NLG)

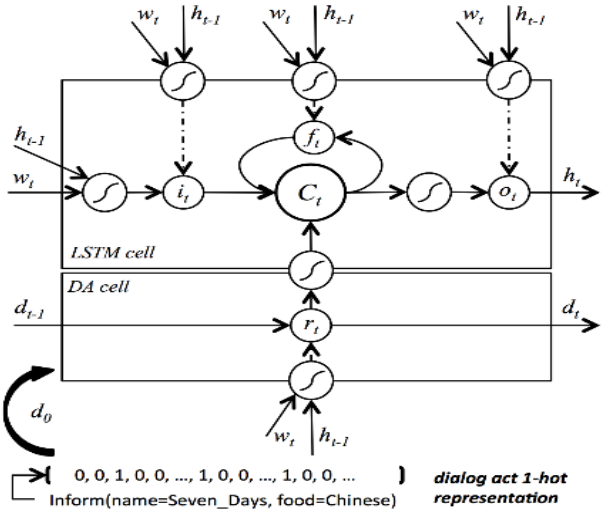
Natural Language Generation (NLG) is the process of generating text from a meaning representation. It can be taken as the reverse of the natural language understanding. NLG systems provide a critical role for text summarization, machine translation, and dialog systems. In the NLG, The system response as a semantic frame, it maps back to a natural language sentence, understandable for the end user. The NLG component can be rule-based or model-based. In some scenarios it can be a hybrid model, i.e. combination of both. The rule-based NLG outputs some predefined template sentences for a given semantic frame, thus they are very limited without any generalisation power. While several general-purpose rule-based generation systems have been developed, they are often quite difficult to adapt to small, task-oriented applications because of their generality. Machine learning based (trainable) NLG systems are more common in today's dialog systems. Such NLG systems use several sources as input such as: content plan, representing meaning representation of what to communicate with the user, knowledge base, structured database to return domain-specific entities, user model, a model that imposes constraints on output utterance, dialog history, the information from previous turns to avoid repetitions, referring expressions, etc.

Trainable NLG systems can produce various candidate utterances (e.g., scholastically or rule base) and use a statistical model to rank them. The statistical model assigns scores to each

utterance and is learnt based on textual data. Most of these systems use bigram and trigram language models to generate utterances.

On the other hand, In NLG based on a semantically controlled Long Short-term Memory (LSTM) recurrent network, It can learn from unaligned data by jointly optimising its sentence planning and surface realisation components using a simple cross entropy training criterion without any heuristics, and good quality language variation is obtained simply by randomly sampling the network outputs.

The following figure shows that the working of Semantic Controlled LSTM cell,

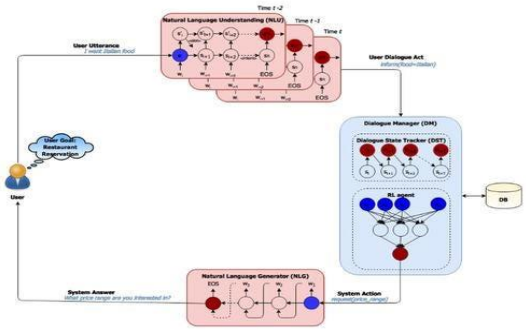


source

Dialogue Management (DM)

The DM could be connected to some external Knowledge Base (KB) or Data Base (DB), such that it can produce more meaningful answers. The Dialogue Manager consists the following two components: the Dialogue State Tracker (DST) and the Policy Learning which is the Reinforcement Learning (RL) agent. The Dialogue State Tracker (DST) is a complex and essential component that should correctly infer the belief about the state of the dialogue, given all the history up to that turn. The Policy Learning is responsible for selecting the best action, i.e. the system response to the user utterance, that should lead the user towards achieving the goal in a minimal number of dialogue turns.

The following figure is shows that how dialogue state Tracker and RL agent are working together,



source

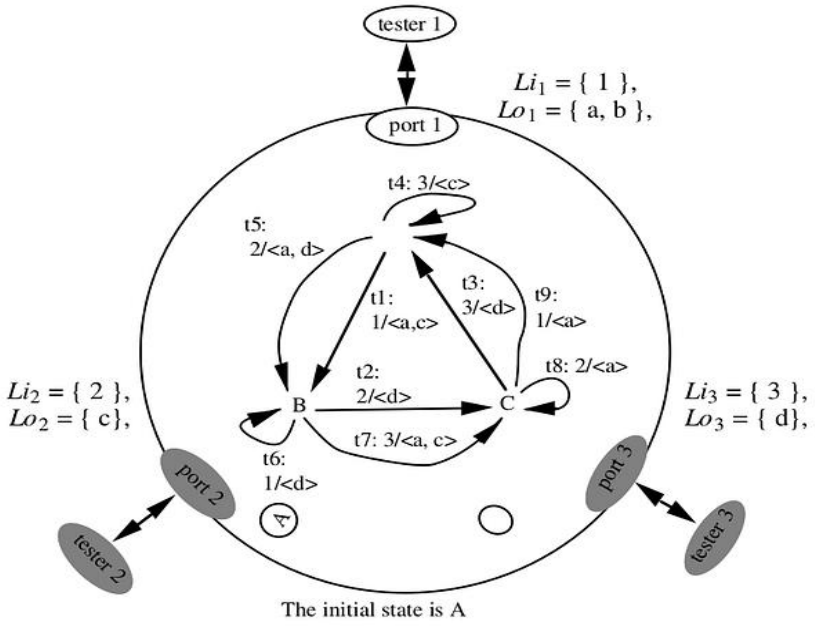
Types of dialog management

I will discuss the different types of dialog management and how they handle these principles.

Finite state machine

The powers of a Finite State Machine are quite extensive. Most conversations can be implemented by a FSM. They are especially good when the number of things a user can say are limited. Most tools for building a conversational bot will also provide a tool to make a decision diagram. So most bots will have a FSM underneath their hood.

A network with distributed terminals sometime can be modelled as a finite state machine with several ports. We define in the following the concept of multi-port finite state machines, which is a generalisation of finite state machines with two ports shows in following figure,



source

Switch statement

The most basic type of dialog management is a large switch statement. Every **intent** triggers a different response. E.g. “Hallo” → “Hi!”, “What’s your name?” → “My name is chatbot”, “What does NLU mean?” → “Natural Language Understanding”, “How are you?” → “I’m doing great!”, etc….

Goal based

In a complex conversation you cannot think about dialogs as a set of states because the number of states can quickly become unmanageable. So you need to approach conversations differently. A popular way of thinking about them is thinking about them in terms of goals.

Say that your user ask for the location of a restaurant without giving it’s name.

i. Your system will receive a “looking_for_restaurant”-intent and start a new goal “finding_restaurant”.

ii. It will notice that to finish this goal it needs to know the name of the restaurant. It therefore will ask the user for the name.

iii. When the user answers it will first analyze this response to see if it contains the name of the restaurant. If it does, it will save the name in its context.

iv. Finally the system will see if it now can finish the “finding_restaurant”-goal. Since the name of the restaurant is now known, it can lookup the restaurant’s location and tell it to the user.

This type of dialog management works based on behaviours instead of states. It’s easier to manage different ways of asking the same question, context switching or making decisions based on what you know about the user.

Belief based

Most NLU will classify intents and entities with a certain degree of uncertainty. This means that dialog manager can only assume what the user said and actually can’t work with discrete rules but needs to work with beliefs.

3. Types of Conversational AI

Rule Based Chatbot

In a rule-based approach, a bot answers questions based on some rules on which it is trained on. The rules defined can be very simple to very complex. The creation of these bots are relatively straightforward using some rule-based approach, but the bot is not efficient in answering questions, whose pattern does not match with the rules on which the bot is trained. However, these systems aren’t able to respond to input patterns or keywords that don’t match existing rules. One of such languages is AIML (Artificial Intelligence Markup Language): The AIML language´s purpose is to make the task of dialog modeling easy, according to the stimulus-response approach. Moreover, it is a XML-based markup language and it is a tag based. Tags are identifiers that are responsible to make code snippets and

insert commands in the chatterbot. AIML defines a data object class called AIML objects, which is responsible for modelling patterns of conversation.

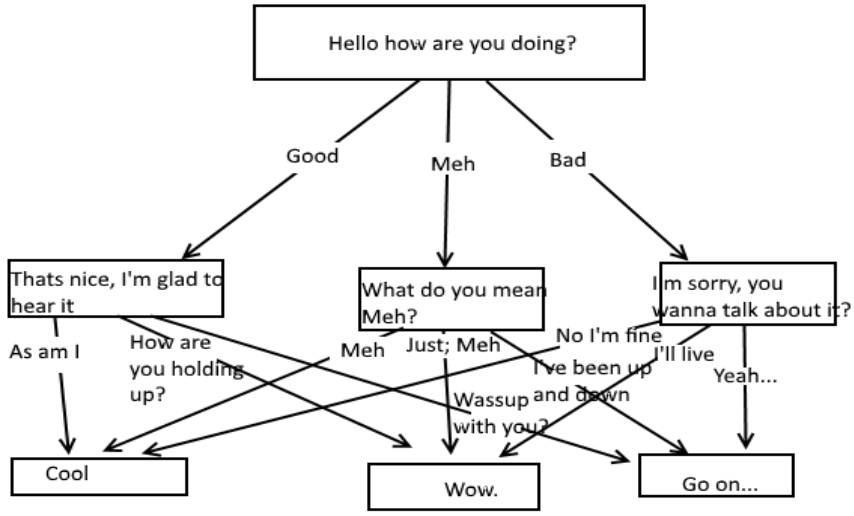
Example of AIML Code,

Basic Tags:

1. **<aiml>:** Defines the beginning and end of an AIML document
2. **<category>:** Defines the knowledge in a knowledge base.
3. **<pattern>:** Defines the pattern to match what a user may input.
4. **<template>:** Defines the response of an Alicebot to user's input.

```
<aiml version="1.0.1" encoding="UTF-8"?>
<category>
      <pattern> HELLO BOT </pattern>
      <template>
      Hello my new friend!
     </template>
</category>
</aiml>
```

The following figure is the Decision tree of rule based conversational AI,

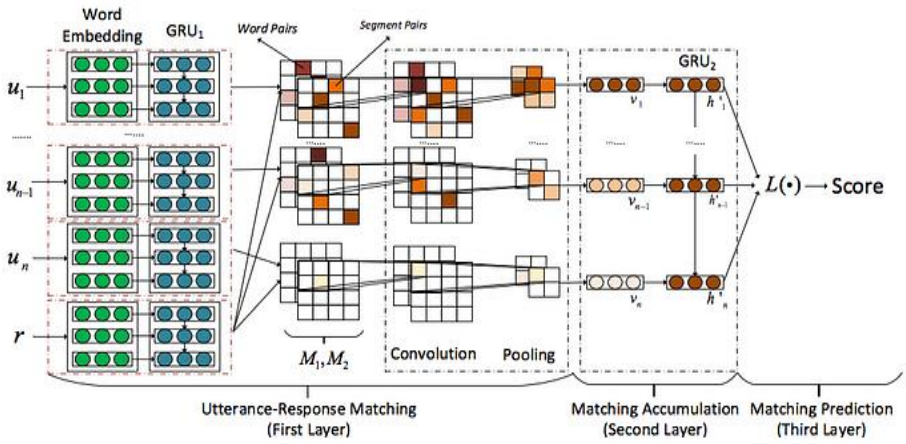


source

Retrieval Based Conversational AI

When given user input, the system uses heuristics to locate the best response from its database of pre-defined responses. Dialogue selection is essentially a prediction problem, and using heuristics to identify the most appropriate response template may involve simple algorithms like keywords matching or it may require more complex processing with machine learning or deep learning. Regardless of the heuristic used, these systems only regurgitate pre-defined responses and do not generate new output.

With massive data available, it is intuitive to build a retrieval based conversational system as information retrieval techniques are developing fast. Given a user input utterance as the query, the system searches for candidate responses by matching metrics. The core of retrieval based conversational systems is formulated as a matching problem between the query utterance and the candidate responses. A typical way for matching is to measure the inner-product of two representing feature vectors for queries and candidate responses in a transformed Hilbert space. The modelling effort boils down to finding the mapping from the original inputs to the feature vectors , which is known as representation learning.There is two-step retrieval technique to find appropriate responses from the massive data repository. The retrieval process consists of a fast ranking by standard **TF-IDF** measurement and the re-ranking process using conversation-oriented features designed with human expertise. The systems to select the most suitable response to the query from the question-answer pairs using a statistical language model as *cross-lingual information retrieval*. These methods are based on shallow representations, which basically utilises one-hot representation of words. Most strong retrieval systems learn representations with deep neural networks (DNNs). DNNs are highly automated learning machines; they can extract underlying abstract features of data automatically by exploring multiple layers of non-linear transformation. Prevailing DNNs for sentence level modelling include convolution neural networks (C-NNs) and recurrent neural networks (RNNs). A series of matching methods can be applied to short-text conversations for retrieval-based systems. Basically, these methods model sentences using convolutional or recurrent networks to construct abstractive representations. Although not all of these methods are originally designed for conversation, they are effective for short-text matching tasks and are included as strong baselines for retrieval-based conversational studies.



source

Response Selection with Topic Clues for Retrieval-based

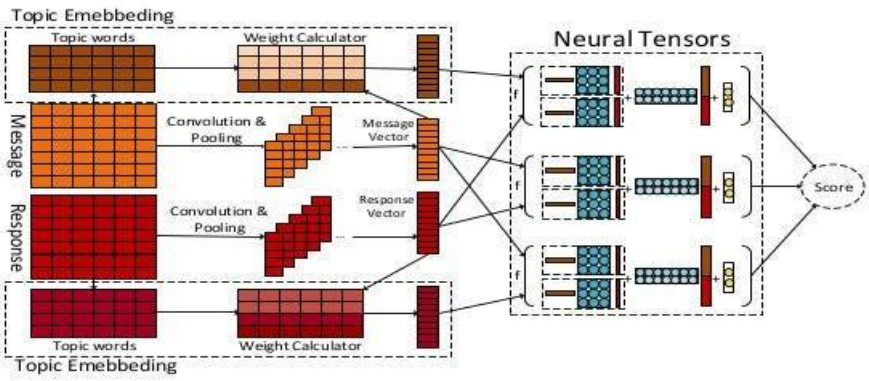
If we have incorporating topic information into message response matching to boost responses with rich content in retrieval-based chatbots.

Topic Word Generation

There is LDA model , which is the state-of-the-art topic model for short texts, to generate topic words for messages and responses. LDA assumes that each piece of text (a message or a response) corresponds to one topic, and each word in the text is either a background word or a topic word under the topic of the text.

Topic-aware Convolutional Neural Tensor Network

There is a topic-aware convolutional neural tensor network (TACNTN) to leverage the topic words obtained from LDA in message-response matching.



source

Generative Based

A generative model chatbot doesn't use any predefined repository. This kind of chatbot is more advanced, because it learns from scratch using a process called "Deep Learning." Generative models are typically based on Machine Translation techniques, but instead of translating from one language to another, we "translate" from an input to an output (response).

Another way to build a conversational system is to use language generation techniques.We can combine language template generation with the search-based methods. With deep learning techniques applied, generation-based systems are greatly advanced.

We have a sequence-to-sequence (seq2seq) framework that emerged in the neural machine translation field and was successfully adapted to dialogue problems. The architecture consists of two RNNs with different sets of parameters.The approach involves two recurrent neural networks, one to encode the source sequence, called the encoder, and a second to decode the encoded source sequence into the target sequence, called the decoder.It was originally developed for machine translation problems, although it has proven successful at related sequence-to-sequence prediction problems such as text summarization and question answering.

Encoder:The encoder simply takes the input data, and train on it then it passes the last state of its recurrent layer as an initial state to the first recurrent layer of the decoder part.

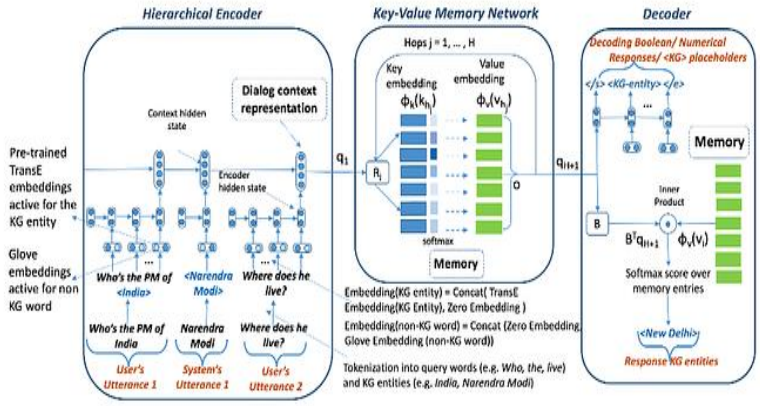
Working of Encoder

The encoder RNN conceives a sequence of context tokens one at a time and updates its hidden state. After processing the whole context sequence, it produces a final hidden state, which incorporates the sense of context and is used for generating the answer.

Decoder: The decoder takes the last state of encoder's last recurrent layer and uses it as an initial state to its first recurrent layer , the input of the decoder is the sequences that we want to get (in our case French sentences).

How Does the Decoder Work?

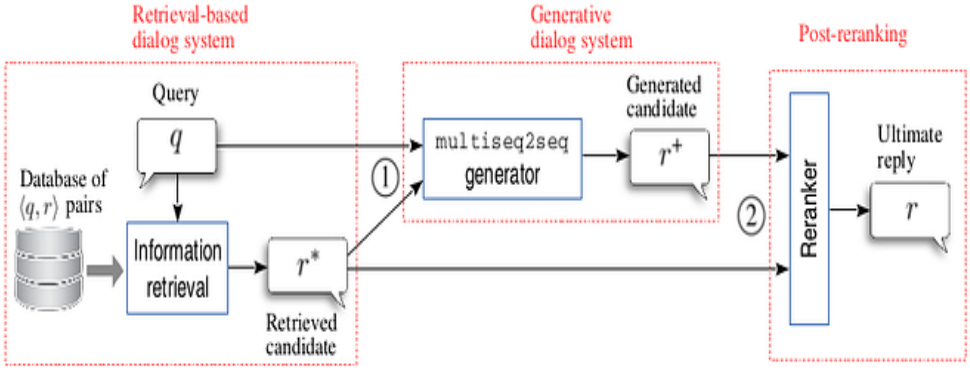
The goal of the decoder is to take context representation from the encoder and generate an answer. For this purpose, a softmax layer over vocabulary is maintained in the decoder RNN. At each time step, this layer takes the decoder hidden state and outputs a probability distribution over all words in its vocabulary.



source

Ensemble of Retrieval- and Generation-Based Dialog Systems

Typically, a recurrent neural network (RNN) captures the query's semantics with one or a few distributed, real-valued vectors (also known as embedding); another RNN decodes the query embedding to a reply. Deep neural networks allow complicated interaction by multiple non-linear transformations; RNNs are further suitable for modelling time-series data (e.g., a sequence of words) especially when enhanced with long short term memory (LSTM) or gated recurrent units (GRUs). Despite these, RNN also has its own weakness when applied to dialog systems: the generated sentence tends to be short, universal, and meaningless, for example, "I don't know" or "something" . This is probably because chatbot-like dialogs are highly diversified and a query may not convey sufficient information for the reply. Even though such universal utterances may be suited in certain dialog context, they make users feel boring and lose interest, and thus are not desirable in real applications.

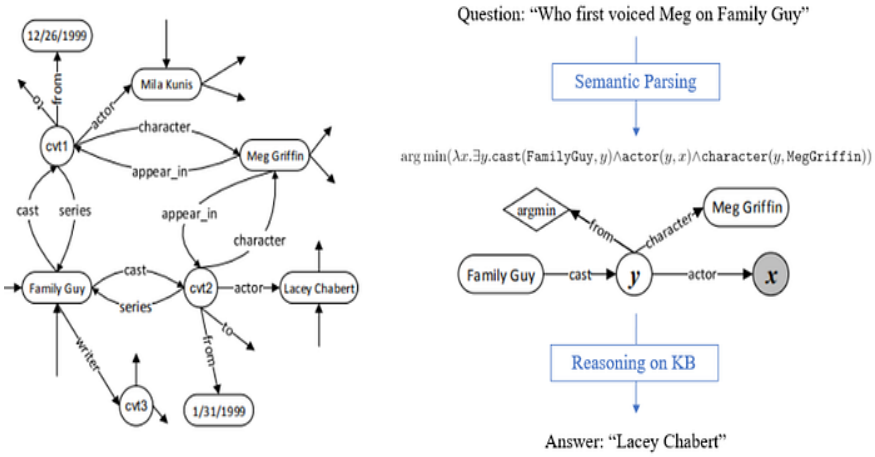


source

AIML Knowledge base (KB) Conversational AI

A KB in this form is often called a Knowledge Graph (KG) due to its graphical representation, i.e., the entities are nodes and the relations the directed edges that link the nodes.

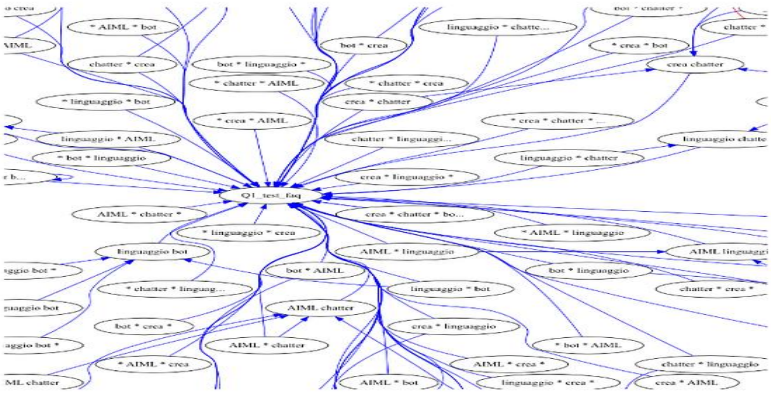
The basic concept of Knowledge base is shown as following figure,



source

Most state-of-the-art symbolic approaches to KB-QA are based on semantic parsing, where a question is mapped to its formal meaning representation (e.g., logical form) and then translated to a KB query. The answers to the question can then be obtained by finding a set of paths in the KB that match the query and retrieving the end nodes of these paths. Knowledge based systems have been helping humans to solve problems which are intellectually difficult, but easy for machines. These problems typically are easily represented with a set of formal rules.

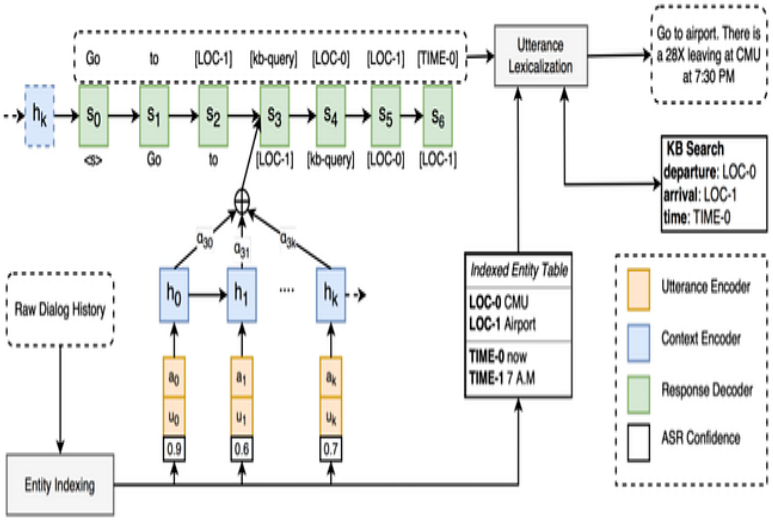
The following figure we have KB representation graph centred on the question Q1. In the graph nodes are patterns (P) and templates (T), and edges are P-T associations and T-P semantic recursions.



source

Knowledge bases (KB) are powerful tools that can be used to augment conversational models. Since knowledge bases usually entail some kind of domain specific information, these techniques are mainly used for task-oriented dialog systems. In a KB, information related to the task at hand can be stored, for example information about nearby restaurants or about public transportation routes. Simple dictionaries or look-up-tables can be used to match an entity with information about it. Since KBs store information discretely, their integration with neural network based encoder-decoder models is not trivial.

The following figure shown that how the KB searching happens,



source

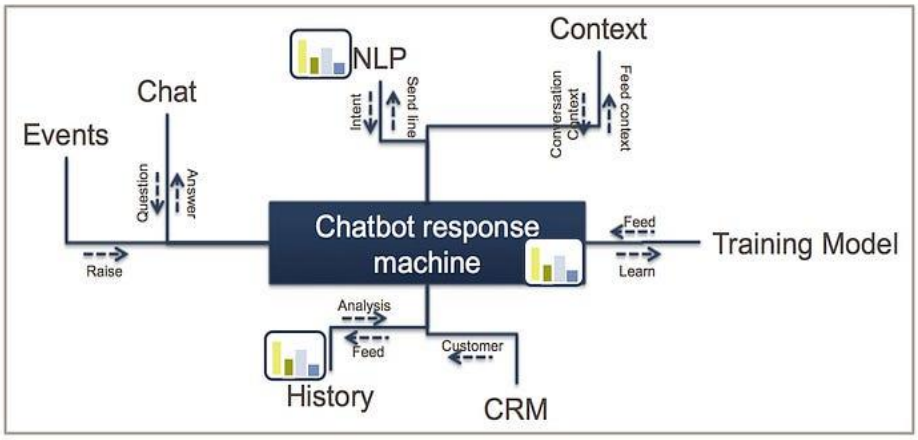
In Restaurant finding Knowledge base mechanism example, the encoder-decoder model produces a response that also uses general tokens for locations and times, and a special placeholder token for the KB result. Finally, the general tokens are transformed back to actual words using the stored table, a KB is employed which uses these general tokens to search for a route between the two places and its output is incorporated in the response. One more similar KB augmented encoder-decoder model is used for the task of recommending restaurants. Here, besides a standard encoder RNN the source utterance is also processed with a belief tracker, implemented as a convolutional neural network (CNN). Convolutional neural networks applied to encoder-decoder models. *Belief tracking* is an important part of task oriented spoken dialog systems. The belief tracker network produces a query for a database containing information about restaurants. The final input to the decoder RNN is the weighted sum consisting of the last state of the decoder RNN and a categorical

probability vector from the belief tracker. Then the decoder outputs a response in the same way as in the previous example, with lexicalised general tokens. These tokens are then replaced with the actual information that they point to in the KB.

Self Learning: Recurrent Embedding Dialogue policy (REDP)

Natural Language Processing with a Training Model to enable the bot to 'learn' to understand a sentence, Context to be able to perform a conversation and History to learn from previous conversations. A grand challenge in this field is to create software which is capable of holding extended conversations, carrying out tasks, keeping track of conversation history, and coherently responding to new information. The aim is to learn vector embeddings for dialogue states and system actions in a supervised setting.

The following figure shown that how the chatbot response machine are associated with all components,



source

When we ask a user "what price range are you looking for?", they might respond with:

- "Why do you need to know that?" (`narrow context`)
- "Can you show me some restaurants yet?" (`broad context`)
- "Actually no I want Chinese food" (`correction`)

- “I should probably cook for myself more” (`chitchat`)

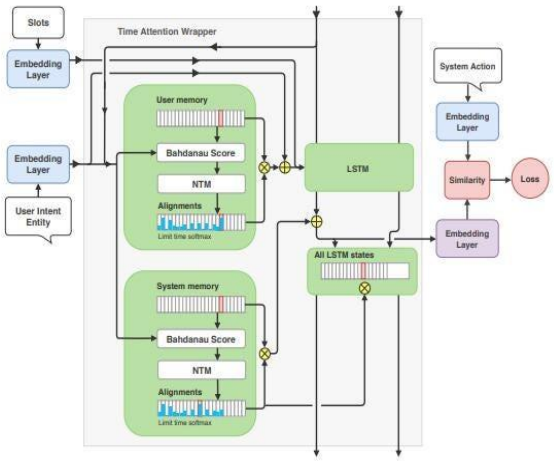
We call all of this *uncooperative* behaviour. There are many other ways a user might respond. Here’s an example conversation:

At inference time, the current state of the dialogue is compared to all possible system actions, and the one with the highest cosine similarity is selected.

REDP, new dialogue policy, has two benefits: (1) it’s much better at learning how to deal with uncooperative behaviour, and (2) it can re-use this information when learning a new task.

It uses the same idea to deal with uncooperative users. After responding correctly to a user’s uncooperative message, the assistant should return to the original task and be able to continue *as though the deviation never happened*. REDP achieves this by adding an attention mechanism to the neural network, allowing it to ignore the irrelevant parts of the dialogue history. The image below is an illustration of the REDP architecture (a full description is in the paper). The attention mechanism is based on a modified version of the Neural Turing Machine, and instead of a classifier we use an embed-and-rank approach.

The following figure shown the working of REDP,

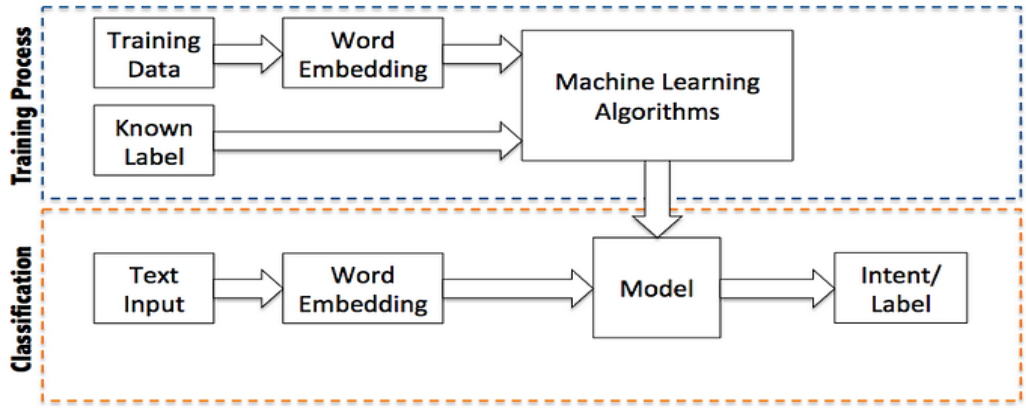


source

Attention has been used in dialogue research before, but the embedding policy is the first model which uses attention specifically for dealing with uncooperative behaviour, and also to reuse that knowledge in a different task. One advantage of this approach is that target labels can be represented as a bag of multiple features, allowing us to represent system actions as a composition of features. In general, the features describing a particular action can come from a number of sources, including the class hierarchy, the name of the action, and even features derived from the code itself (such as which functions are called). Whatever the source of the features, similar actions should have more features in common than dissimilar actions, and ideally reflect the structure of the domains. In our experiments we only derive features from the action name, either taking the whole name as a single feature, or splitting the name into tokens and representing it as a bag of words.

4. Intent Identification and Information Extraction

The machine algorithm for Intent Identification can be either supervised or unsupervised. If we implement the supervised approach, we need to manually give labels to hundreds of data for training purpose which going to be tiring and boring, but if we implement the unsupervised one, there were several critical knowledge gaps that we can't cover in just 3 weeks especially regarding the design of training process. Therefore, even though we need to manually give labels to our data, we chose to go with the supervised one. This picture below illustrate how text classifier using supervised ML works:



source

At this point, we have several machine learning algorithms that we could choose to implement supervised learning which is Naive Bayesian, LDA, SVM and Neural Network. But

before we choose the algorithm, we need to find a method to translate words into an array (word embedding) since all algorithms that I mention previously need input in form of array or at least numbers. There are 2 options that we had to do that, by using one hot encoded bag of words (bow) or word2vec (CBOW). If we had more times, we definitely would choose word2vec to embed the input since the size of array would be significantly smaller compared to BOW, but we had limited time and to implement word2vec we need to use Java (deeplearn4j) or Python(gensim) which no one between us had any experience making an API using these languages. Actually, it is possible for us to create the classifier by using Python but the problem will occur in the process of making an API out of it, especially in the deployment process. To deploy Python in the live server, there are several configurations that need to be done and we don't have the courage to play around with our company server since everyone else is also using it for other projects. So for the sake of familiarity, we decide to use BOW which we manage to find a node package to implement it called mimir.

Regarding Information Extraction, The primary responsibility of the NLU is not just to understand phrase function, but to understand the meaning of the text itself. To extract meaning from text, we convert unstructured text — text written into a text-only chatbot — into structured grammatical data objects, which will be further processed by the Dialogue Manager. The first step in this process is breaking down a sentence into tokens that represent each of its component parts: words, punctuation marks, numbers, etc. Tokenization is difficult because of the frequency of ambiguous or malformed inputs including: (i) phrases , (ii) contractions , abbreviations , and periods. These tokens can be analyzed using a number of techniques, described below, to create a number of different data structures that be processed by the dialogue manager.

There are few approach which can be use for Information retrieval as below,

Bag of Words: We ignore sentence structure, order, and syntax, and count the number of occurrences of each word. We use this to form a vector space model, in which stop words are removed, and morphological variants go through a process call lemmatization and are stored as instances of the basic lemma . In the dialogue manager phase, assuming a rule-based bot, these resulting words will be matched against documents stored in the bot's knowledge database to find the documents with inputs containing similar keywords. The bag of words

approach is simple because it does not require knowledge of syntax, but, for this same reason, is not precise enough to solve more complex problems.

Latent Semantic Analysis : This approach is similar to the bag of words. Meanings / concepts, however, not words, are the basic unit of comparison parsed from a given sentence or utterance. Second, groups of words that co-occur frequently are grouped together. In LSA, we create a matrix where each row represents a unique word, each column represents a document, and the value of each cell is the frequency of the word in the document. We compute the distance between the vector representing each utterance and document, using singular value decomposition to reduce the dimensionality of the matrix, and determine the closest document.

Regular Expressions: Sentences / utterances can be treated as regular expressions, and can be pattern matched against the documents in the bot's knowledge database. For example, imagine that one of the documents in the bot's knowledge database handles the case where the user inputs the phrase: "my name is *". "*" is the wildcard character, and indicates that this regular expression should be triggered whenever the bot hears the phrase "my name is" followed by anything. If the user says "my name is Jack", this phrase will be parsed into a number of regular expressions, including "my name is *" and will trigger the retrieval of that document.

Part of Speech (POS) Tagging: POS tagging labels each word in the input string with its part of speech (e.g. noun, verb, adjective, etc.). These labels can be rule-based (a manually-created set of rules is created to specify part of speech for ambiguous words given their context). They can also be created using stochastic models which train on sentences labeled with correct POS. In the dialogue manager, POS can be used to store relevant information in the dialogue history. POS is also used in response generation to indicate the POS object type of the desired response.

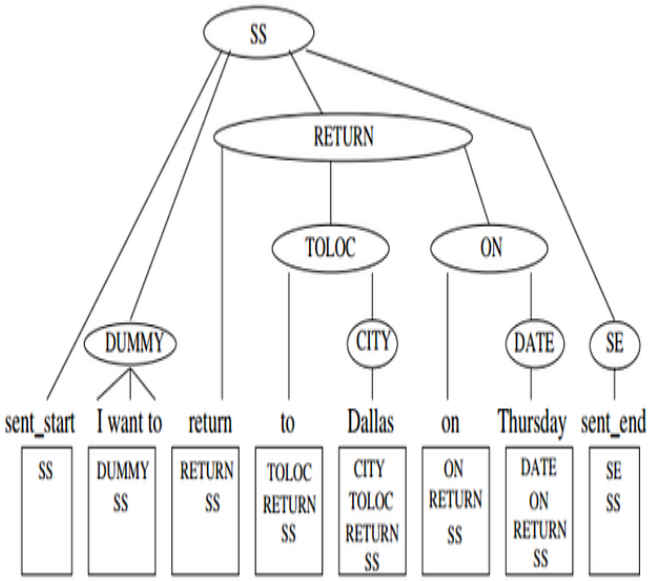
Named/Relation Entity Recognition: In named entity recognition (NER), the names of people, places, groups, and locations are extracted and labeled accordingly. NER-name pairs can be stored by the dialogue manager in the dialogue history to keep track of the context of the bot's conversation. Relation extraction goes one step further to identity relations (e.g. "who did what to whom") and label each word in these phrases.

Semantic Role Labelling: The arguments of a verb are labelled based on their semantic role (e.g. subject, theme, etc.). In this process, the predicate is labelled first followed by its arguments. Prominent classifiers for semantic role labelling have been trained on FrameNet and PropBank, databases with sentences already labelled with their semantic roles. These semantic role-word pairs can be stored by the dialogue manager in the dialogue history to keep track of context.

Creation of Grammatical Data Structures: Sentences and utterances can be stored in a structured way in grammar formalism such as context-free grammars (CFGs) and dependency grammars (DGs). Context-free grammars are tree-like data structures that represent sentences as containing noun phrases and verb phrases, each of which contain nouns, verbs, subjects, and other grammatical constructs. Dependency grammars, by contrast, focus on the relationships between words.

Statistical Methods for Information Extraction

Hidden Vector State (HVS) Model: The goal of the statistical hidden vector state models is to automatically produce some accurate structured meaning. Consider an example as “I want to return to Dallas on Thursday.” The parse tree below represents one way of representing the structured meaning of the sentence. SS represents the initial node send_start, and SE represents the end node send_end. We view each leaf node as a vector state, described by its parent nodes: the vector state of Dallas is [CITY, TOLOC, RETURN, SS].



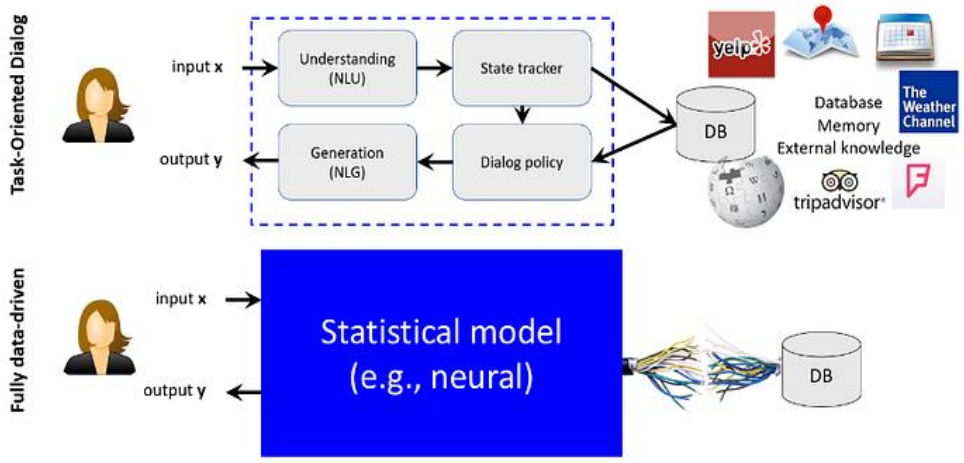
source

The whole parse-tree can then be thought of a sequence of vector states, represented by the sequence of squares above. If each vector state is thought of as a hidden variable, then the sequence of vector states (e.g. squares above) can be thought of as a Hidden Markov Model: we start at SS, and have certain probabilities of reaching a number of possible hidden states as the next state. Each vector state can be thought of as a "push-down automaton" or stack. **Support Vector Machine (SVM) Model:** Support Vector Machines are a supervised machine learning tool. Given a set of labeled training data, the algorithm generates the optimal hyperplane that divides the sample into their proper labels. Traditionally, SVMs are thought of as solving binary classification problems, however multiple hyperplanes can be used to divide the data into more than two label categories. The optimal hyperplane is defined as the hyperplane that creates the maximum margin, or distance, between different-labeled data point sets.

Conditional Random Field Models: CRFs are log-linear statistical models often applied for structured prediction. Unlike the average classifier, which predicts a label for a single object and ignores context, CRF's take into account previous features of the input sequence through the use of conditional probabilities. A number of different features can be used to train the model, including lexical information, prefixes and suffixes, capitalization and other features.

Deep Learning: The most recent advancement in the use of statistical models for concept structure prediction is deep learning for natural language processing, or deep NLP. Deep learning neural network architectures differ from traditional neural networks in that they use more hidden layers, with each layer handling increasingly complex features. As a result, the networks can learn from patterns and unlabelled data, and deep learning can be used for unsupervised learning. Deep learning methods have been used to generate POS tags of sentences (chunk text into noun phrases, verb phrases, etc.) and for named-entity recognition and semantic role labelling.

The below picture illustrate the working of Deep learning based **Statistical Model,**



source

5. Self-learning Based on Sentiment Analysis

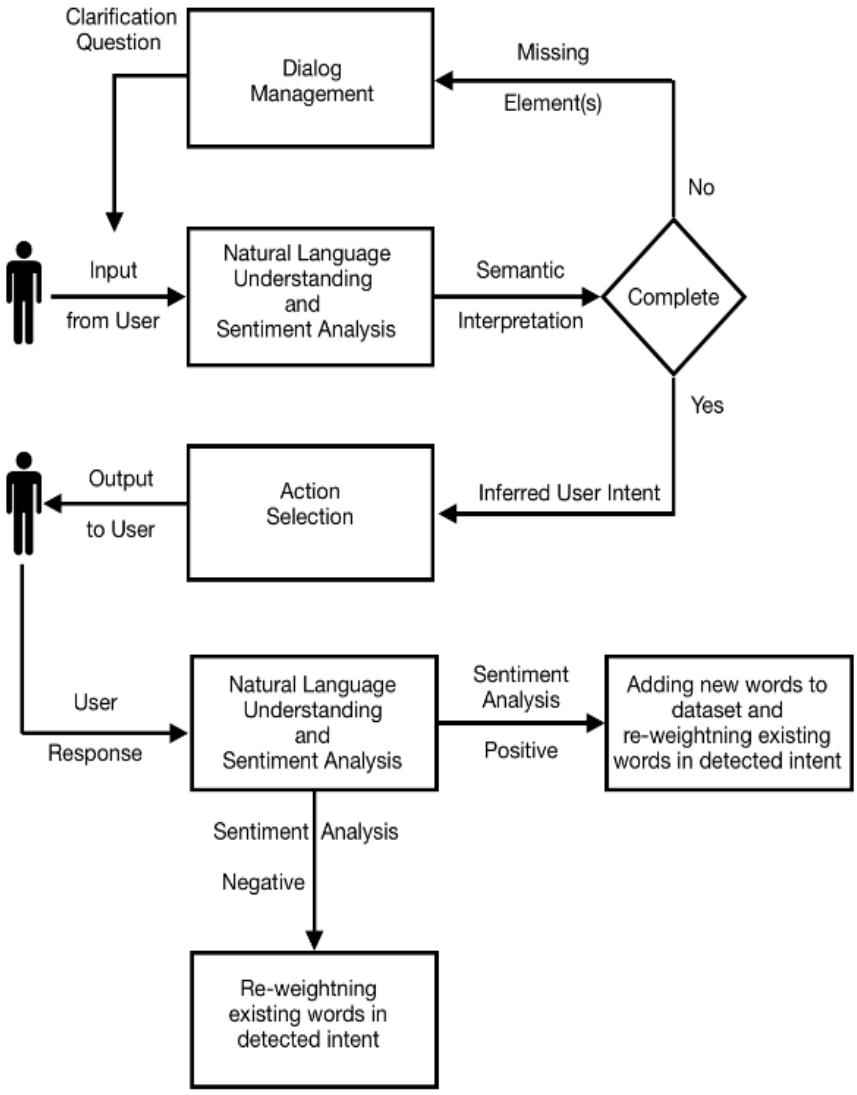
Initially, the development of a bot was based on two fundamental components :

Natural Language Understanding module, used by the Dialogue Manager, that processes the user input to search for keywords through which to understand the action to be taken.

Natural Language Generation module that generates answers from the information gathered by the Dialogue Manager.

Over time, we have faced a real evolution in the development of task-oriented conversational agents because of the availability of deep learning techniques.

This picture below illustrate the process of sentiment analysis in user generated content,



source

The training process for sentiment analysis it will provide for automatic labeling of new instances. In the sentiment analysis method each sentence is analyzed against two classification sub-systems: one for identifying the class of the answers, one for assessing the sentiment of the sentence. At the end of the processing of each sentence the learning model is updated according to the detected sentiment. This is based on a data structure formed by

intents (An intent is a semantic label representing an intention of the end-user) . For each intent, there is a set of sentences that represent it. Each sentence that describes an intent contains entities (Entities are the parameters of the intent that help in defining the specific user request) that are attributes specific to the given intent.

Conclusion:

As we conclude our exploration of chatbots, it becomes evident that these conversational agents represent a pivotal technological shift in human-computer interaction. Their influence extends far beyond the realm of novelty, offering tangible advantages across various domains. Chatbots bring to the table the virtues of round-the-clock availability, operational efficiency, cost-effectiveness, and unwavering consistency in user engagement. Despite the persisting challenges related to their capacity for understanding complex inputs and the intricacies of seamless integration, chatbots are on a trajectory of continuous improvement.

The trajectory of chatbots is a journey still very much in progress. Their potential for innovation and integration into industries such as healthcare, education, finance, and beyond is vast, and both businesses and individuals continue to explore and harness the advantages of these AI-driven conversational interfaces. Chatbots are not just lines of code; they signify a profound transformation in how we interact with technology and information. As the underlying AI technologies mature and algorithms become more sophisticated, the future holds promise for even more remarkable developments in the chatbot landscape. With every interaction, they are reshaping the landscape of digital engagement, making it clear that chatbots are here to stay and will play a pivotal role in shaping our digital future.