

**GOVERNMENT COLLEGE OF ENGINEERING BARGUR**

**( AUTONOMOUS)**

**PROJECT TITLE: CHATBOT DEPLOYMENT WITH IBM CLOUD WATSON ASSISTANT**

**TEAM MEMBERS:**

**DHINOOVIKA D**

**HARSHANA R**

**KIRANA B**

**PAVITHRA K**

**VISALI B**

**PROBLEM STATEMENT:**

 Create a helpful virtual guide using IBM Cloud Watson Assistant. Customize the chatbot to assist users on popular messaging platforms like Facebook Messenger and Slack. Provide useful information, answer FAQs, and offer a friendly conversational experience. Empower users with quick access to information and create meaningful connections through your virtual guide!

**PROBLEM SOLUTION :**

1. Persona Design: Define the chatbot's persona, including its name, tone, and style of communication.

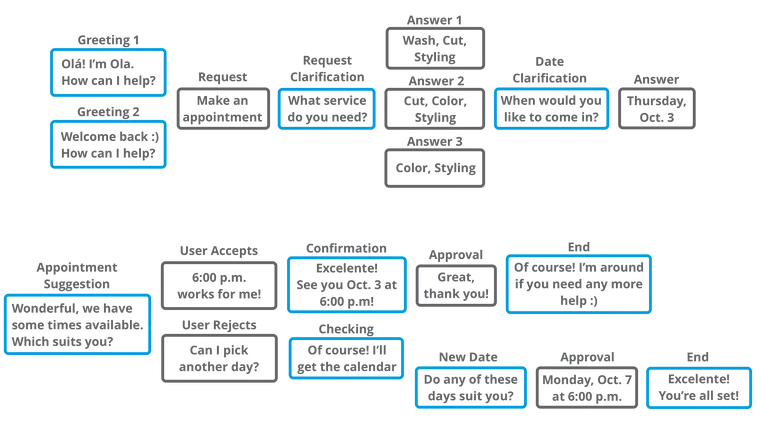
NAME OF THE CHATBOT :**ZENOBOT**

TONE :**FORMAL**

STYLE OF COMMUNICATION :**FRIENDLY**

1. User Scenarios: Identify common user scenarios and FAQs that the chatbot should be able to address.

Example scenario:



Example FAQ:

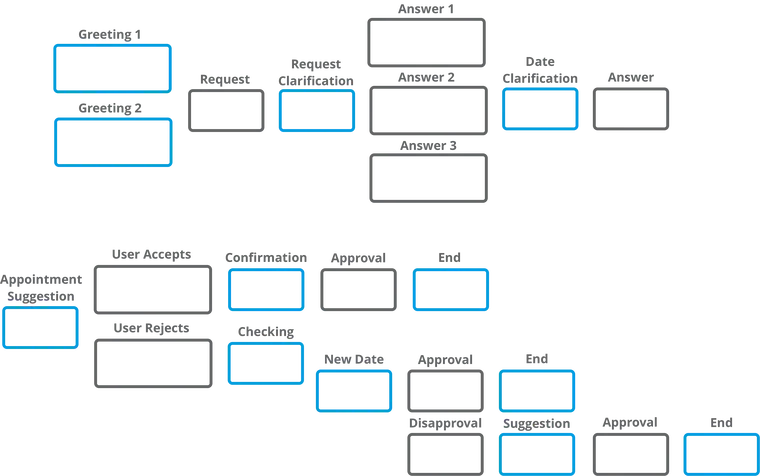
1. Conversation Flow: Design the conversation flow, outlining how the chatbot responds to user queries and prompts.

Write your script in fragments to stay organized and make brainstorming easier. An easy way to do this is by creating a [conversation diagram.](https://miro.medium.com/max/3116/1*Y5DokuM8QuKXaJdz12WtOw.png)

Conversations have elements, and a diagram will help you map out every possibility of what your chatbot could say. The elements you may use in a chatbot conversation are:

* **Greeting:**Used to say hello or start a conversation. Formality is dependent on relationship (return versus new users).
* **Asking:**For engaging or seeking information. Helps keep the conversation going.
* **Informing:**Giving information that is either requested or pertinent to the conversation.
* **Checking:**Testing the user’s understanding. Restating details and information for clarity.
* **Error:**When the chatbot doesn’t understand or fails to fulfill a request.
* **Apologizing:**Politely acknowledging the chatbot’s shortcomings. Should be brief and serve as a bridge to alternative solutions.
* **Suggesting:**Presents the user with relevant actions or options.
* **Conclusion:**A clear end to the conversation.

Visual elements count as well such as GIFs, emoji, pictures or videos.



1. Response Configuration: Configure the chatbot's responses using Watson Assistant's intents, entities, and dialog nodes

**ADD INTENTS:**

For example, in a discussion with the support team, you might gather this set of standard questions that support received from users:

* What is the status of the business application? I could not access it.
* How to get access to a business application?
* How to reset my password for a specific application?
* When to renew my workstation?
* How to bring my own device and connect it to enterprise network?

Each of those questions is documented as a frequently asked question in the support team's document repository. Some solutions persist in a relational database in the form of application > problem > solution.

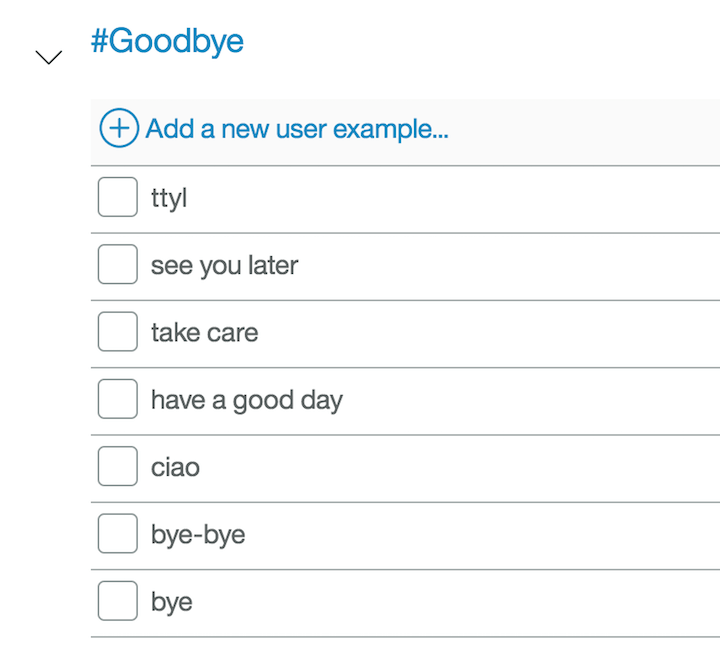
Based on the questions, you can extract these intents:

* Access to a business applications like expense report, AbC.
* Reset password
* Access to supplier on boarding business process
* Bring your own device

Add those intents to the workspace: From the Build page, click **Intents** and click **Create new**.

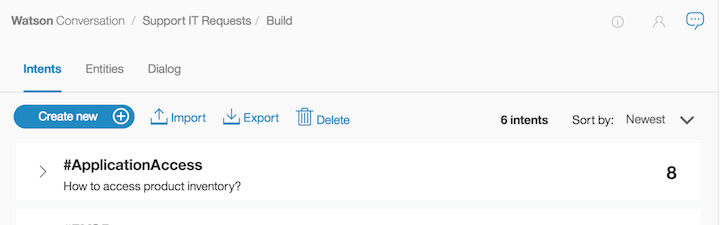
For the intent name, type applicationAccess after the number sign (#).

For each intent, add examples to train the conversation for intent recognition.

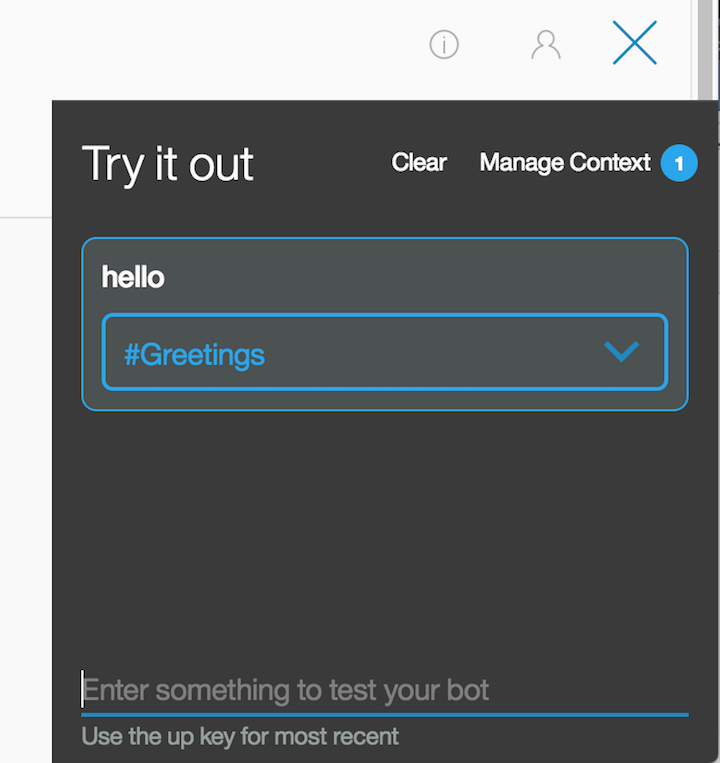
Create the Goodbyes intent and add examples for it. Because many intents can be reused from conversation to conversation implementations, you can define .csv files and import them in the Conversation Tool Intents. The .csv format is shown in this example with one intent per line: To get the IT support demonstration intents, click the **Import** link on the Intents page to import the wcs-workspace/ITSupport-Intents.csv file from the GitHub repository.

Next, test your conversations.

1. As soon as you create an intent, you can test it by clicking **Ask Watson** icon in the top, right-hand side of the conversation editor.



Enter one of the examples. You should get the #greetings intent identified by Watson. Enter other greetings to test the #greetings intent.



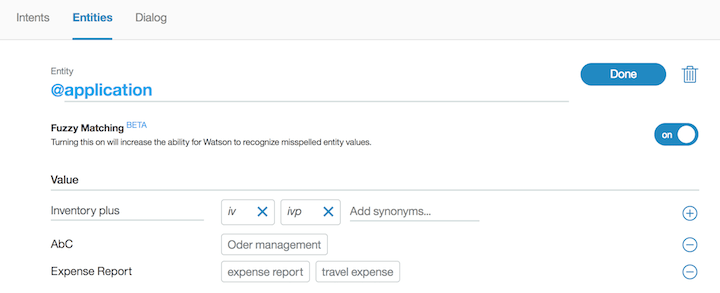
**ADD ENTITIES:**

Click **Entities**. On the Entities page, click **Create new**.

1. Adding values and synonyms to entities helps your chatbot learn important details that your users might mention.

Each entity definition includes a set of specific entity values that can be used to trigger different responses. Each value can have multiple synonyms that define different ways that the same value can be specified in user input.

1. Create entities to represent to the application what the user wants to access.



*Fuzzy logic* is a feature that allows Watson Assistant to accept misspelled words. You can enable this feature at the entity level.

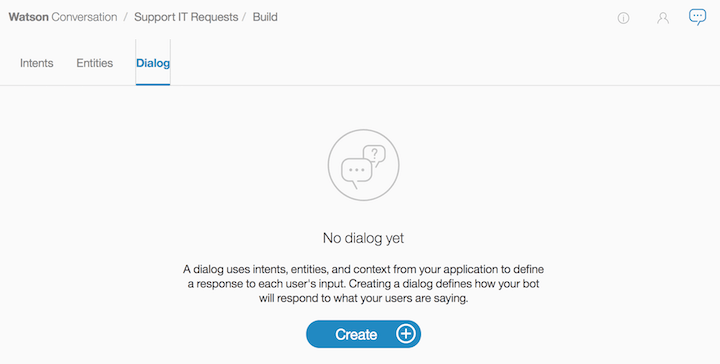
As you did for intents, you can reuse entities' definitions through the export and import capabilities. Import the wcs-workspace/ITSupport-Entities.csv file.

1. If you click the **Ask Watson** icon immediately after you import the entities, the Watson is training message is displayed. Watson Assistant classifies the entities. You can unit-test the entities by entering I want to access application AbC. The following figure shows both the intent and entity (@application:AbC) extracted by Watson Assistant:

You are now ready to create the dialog flow.

**ADD DIALOG FLOW:**

After you specify your intents and entities, you can construct the dialog flow.



A dialog is made up of nodes that define steps in the conversation.

The first node is the standard welcome message. The other node is a catch-all node named "Anything else." Dialog nodes are chained in a tree structure to create an interactive conversation with the user. The evaluation starts at the top, so the welcome node is assessed before the "Anything else" node.

If you click the welcome node, the standard Watson response is "Hello. How can I help you?" To validate how the flow works, you can click the **Ask Watson** icon.

1. Platform Integration: Integrate the chatbot with popular messaging platforms like Facebook Messenger and Slack.

**FACEBOOK INTEGRATION:**

1. Go to the **Integrations** page by clicking the integrations icon (Integrations icon) in the left menu.
2. Click **Add** on the **Facebook Messenger** tile.
3. Click **Confirm**.
4. Follow the instructions that are provided on the screen to complete the integration process.

**Action considerations**

The rich responses that you add to the action are displayed in a Facebook app as expected, with the following exceptions:

* **Connect to live agent**: This response type is ignored.
* **Image**: This response type embeds an image in the response. A title and description are not displayed before the image, whether or not you specify them.
* **Option**: This response type shows a list of options that the user can choose from.
  + A description is not displayed, whether you specify one or not.
  + After a user clicks one of the buttons, the button choices disappear and are replaced by the user input that is generated by the user's choice. If the assistant or the user enters new input, then the button-generated input disappears. Therefore, if you include multiple response types in a single response, position the option response type last. Otherwise, content from subsequent responses, such as text from a text response type, will replace the button-generated text.
  + The title is automatically taken from the text of the relevant step of the action where options are listed.

**SLACK INTEGRATION:**

1. Go to the **Integrations** page by clicking the integrations icon (Integrations icon) in the left menu.
2. Click **Add** on the **Slack** tile.
3. Click **Confirm**.
4. You need to have a Slack app to connect to.

If you don’t have a Slack app, create one now. See [Starting with Slack apps](https://api.slack.com/start).

1. Go to the [Your Apps](https://api.slack.com/apps) page on the Slack website, and then click the app you want to use.
2. From the settings page for your Slack app, open the **App Home** page.
3. Add access scopes for your Slack app.
4. Assign bot token scopes to your Slack app. At a minimum, apply the following scopes:
   * app\_mentions:read
   * chat:write
   * im:history
   * im:read
   * im:write
5. Click *Install App to Workspace*, and then allow the installation when prompted.

If you are editing scopes for an existing application, reinstall it.

1. From the Slack settings App Home page, enable the *Always Show My Bot As Online* setting.
2. Go to the *OAuth and Permissions* page in Slack, copy the *Bot User OAuth Access Token*.
3. From the Watson Assistant Slack integration configuration page, paste the token that you copied in the previous step into both the **OAuth access token** and **Bot user OAuth access token** fields.
4. On the Slack app settings page, go to the *Basic Information* page, and then find the *App Credentials* section. Copy the app credential verification token.
5. From the Watson Assistant Slack integration configuration page, paste the verification token that you copied in the previous step into the **Verification token** field.
6. Click **Generate request URL**, and then copy the generated request URL.
7. Return to the Slack app settings page. Open the *Event Subscriptions* page, and then turn on *Enable Events*. Paste the request URL that you copied in the previous step into the field.
8. On the *Event Subscriptions* page in Slack, find the *Subscribe to Bot Events* section. Click *Add Bot User Event*, and then select the event types you want to subscribe to. You must select at least one of the following types:
   * message.im: Listens for message events that are posted in a direct message channel.
   * app\_mention: Listens for only message events that mention your app or bot.
9. Click *Save Changes*.
10. Optional: To add support for showing buttons, menus, and disambiguation options in the Slack app, go to the *Interactive Components* tab and enable the feature. Paste your request URL in the provided text entry field, and then click *Enable Interactive Components*.

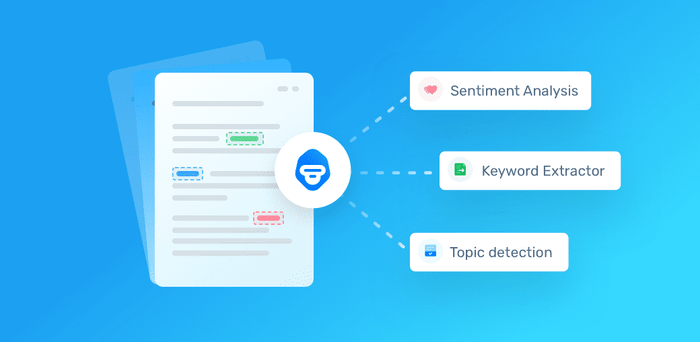
Chatbots have become an essential tool for businesses to enhance customer interaction and streamline processes. With the advancement of technology, integrating Natural Language Understanding (NLU) into chatbots has become crucial for accurate user intent recognition and providing a seamless user experience. This document explores the implementation of NLU in IBM Cloud Watson Assistant, focusing on improving user interaction and problem-solving capabilities.

ZENOBOT



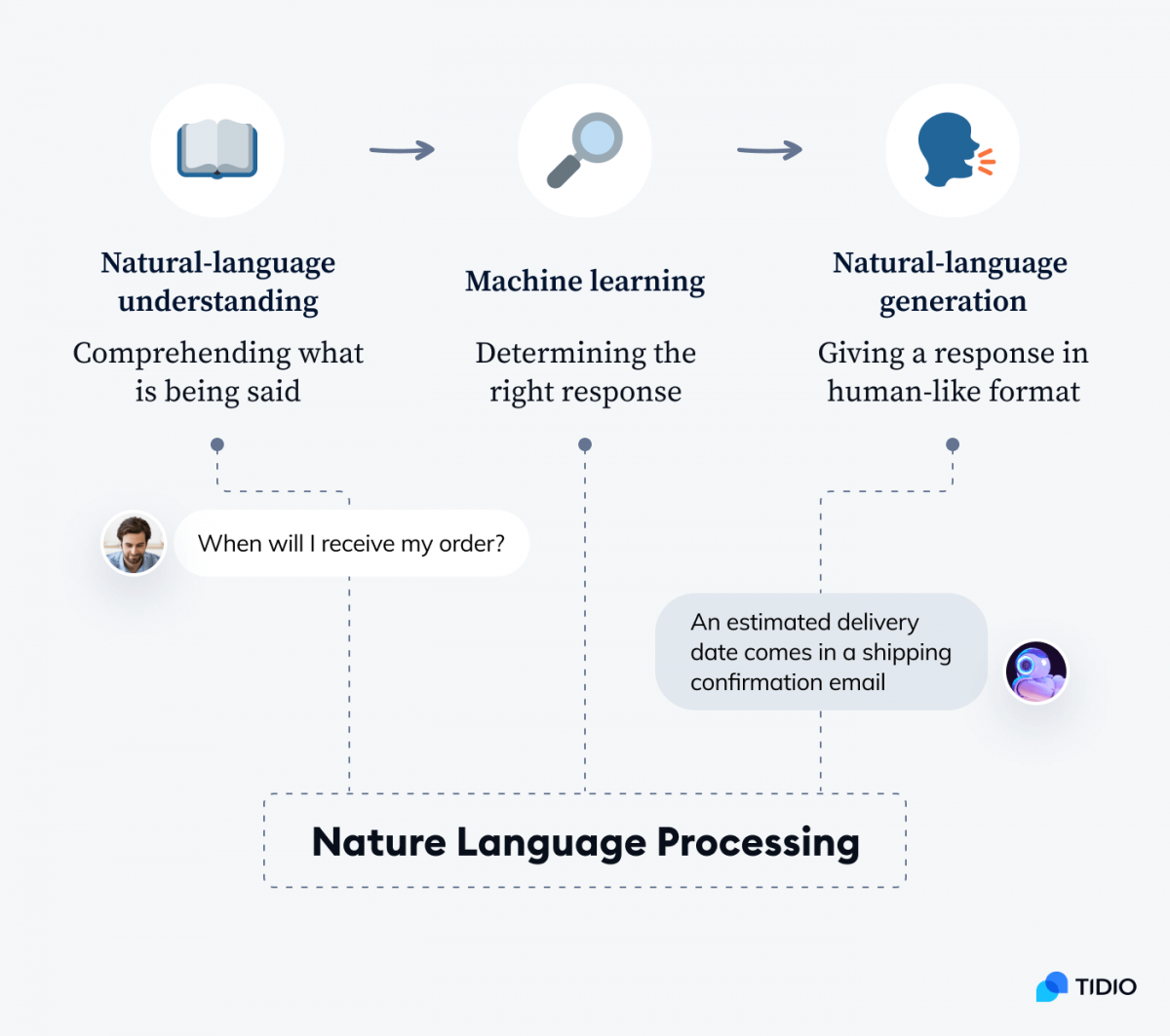
1.Understanding Natural Language Understanding (NLU):

NLU is a branch of artificial intelligence that focuses on the interaction between humans and computers using natural language. It enables chatbots to comprehend user inputs, extract meaningful information, and respond contextually. In the context of IBM Cloud Watson Assistant, NLU enhances the chatbot’s ability to interpret user queries accurately.



2.Benefits of Implementing NLU in Chatbots:

Improved User Intent Recognition: NLU helps the chatbot understand user intents with higher accuracy, leading to precise responses. Enhanced Context Awareness: NLU enables the chatbot to understand the context of the conversation, making interactions more personalized. 3 Efficient Problem-Solving: With accurate intent recognition, chatbots equipped with NLU can efficiently address user queries and solve problems.



3. Steps to Implement NLU in IBM Cloud Watson Assistant:

a. Configure Language Models: Set up language models to recognize various languages and dialects used by your target audience. b. Train NLU Models: Train the NLU models using relevant datasets to improve the accuracy of intent recognition. c. Integrate with Watson Assistant: Integrate the trained NLU models seamlessly with IBM Cloud Watson Assistant. d. Test and Iterate: Conduct rigorous testing to ensure the chatbot accurately interprets user intents. Iterate on the training data and models as needed to enhance accuracy

**Building the project by loading and preprocessing the database:**

1. Import the libraries:

import tensorflow

import nltk

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout , Activation, Flatten , Conv2D, MaxPooling2D

from tensorflow.keras.optimizers import SGD

import random

import json

import pickle

These all are the libraries we will require to create our project. nltk is required to tokenize words and sentences and also to lemmatize words. Basically nltk will be required to preprocess our data(we have to perform certain operations on our data as we will be working on a json file which contains words, and sentences).

And we will be using Tensorflow for creating our model, numpy to convert our data into array form. Random to generate random responses according to the user message. Json to read the json file. Pickle to save our labels and words.

2. Declaring Constants:

In this step, we declare some constants that will be required while separating sentences.

words=[]

labels = []

docs = []

ignore\_list = ['?', '!']

3. Loading our dataset that is intents.json file:

Load the json dataset using json.loads() method.

dataset = open('intents.json').read()

intents = json.loads(dataset)

4. Preprocess Data:

**for** intent **in** intents['intents']:

**for** pattern **in** intent['patterns']:

#tokenize each word

word\_token = nltk.word\_tokenize(pattern)

words.extend(word\_token)

#add documents in the corpus

docs.append((word\_token, intent['tag']))

# add to our labels list

**if** intent['tag'] not **in** labels:

labels.append(intent['tag'])

As we are working on text data, we need to perform certain operations or say preprocessing on data before creating a model to train on that data. So, in the above code we first iterate through our intents and patterns, and we tokenize each sentence present in that pattern (Tokenizing means breaking text into small parts like words), and then append each tokenize word into the words list. And in this we also create a list of labels for our tags.

5. Lemmatizing Each word:

# lemmatize each word, and sort words by removing duplicates:

words = [lemmatizer.lemmatize(word.lower()) **for** word **in** words **if** word not **in** ignore\_list]

words = sorted(list(set(words)))

# sort labels:

labels = sorted(list(set(labels)))

In this code, we lemmatize each word (Lemmatizing means converting a word into its lemma form) and also remove duplicate words from the list and sort words and labels list.

6. Save words and labels list (using pickle):

Now we will save our words and labels list that we have created using the pickle library.

pickle.dump(words,open('words.pkl','wb'))

pickle.dump(labels,open('labels.pkl','wb'))

7. Creating our Training data:

# creating our training data:

training\_data = []

# creating an empty array for our output (with size same as length of labels):

output = [0]\*len(labels)

**for** doc **in** docs:

bag\_of\_words = []

pattern\_words = doc[0]

#lemmatize pattern words:

pattern\_words = [lemmatizer.lemmatize(word.lower()) **for** word **in** pattern\_words]

**for** w **in** words:

**if** w **in** pattern\_words:

bag\_of\_words.append(1)

**else**:

bag\_of\_words.append(0)

output\_row = list(output)

output\_row[labels.index(doc[1])] = 1

training\_data.append([bag\_of\_words,output\_row])

In this piece of code, we create our training data in which we will provide the input that is bag\_of\_words that will be pattern and ouput\_row which will be the output which tells us in which label our pattern belongs to. As the computer doesn’t understand text, that is why we have converted it to numbers.

8. Shuffle and Convert our Training data to array:

We shuffle our training data using random.shuffle() method, and also convert our data to a numpy array using numpy library.

# convert training\_data to numpy array and shuffle the data:

random.shuffle(training\_data)

training\_data = np.array(training\_data)

9. Splitting the data into x\_train and y\_train:

Splitting our training data into x\_train and y\_train. X\_train consist of words and y\_train consists of its corresponding label.

# Now we have to create training list:

x\_train = list(training\_data[:,0])

y\_train = list(training\_data[:,1])

10. Model Creation:

# Creating Model:

model = Sequential()

model.add(Dense(128, input\_shape=(len(x\_train[0]),), activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(len(y\_train[0]), activation='softmax'))

In this model, we will create 3 fully connected layers in which there is one input layer and one output layer. As you can see in the above code.

11. Compile and Fit our model to find the accuracy:

sgd\_optimizer = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=**True**)

model.compile(loss='categorical\_crossentropy', optimizer=sgd\_optimizer, metrics=['accuracy'])

In this, we will be using a Stochastic gradient descent(sgd) optimizer with Nesterov accelerated gradient.

# fit the model

history = model.fit(np.array(x\_train), np.array(y\_train), epochs=200, batch\_size=5, verbose=1)

12. Save the model:

Now after creating the model we will save our model using save() method.

model.save('chatbot\_Application\_model.h5', history)

13. Final step to predict the sentences and get responses:

Now, we have to create one more python file in which we load our model, we load our words list, labels list that we have saved above. As we know that our model will only predict the label in which it belongs to, so we have to create certain functions which will identify the label and provide random responses from the list of responses.

import nltk

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

import pickle

import numpy as np

import json

import random

from keras.models import load\_model

model = load\_model('chatbot\_Application\_model.h5')

intents = json.loads(open('intents.json').read())

words = pickle.load(open('words.pkl','rb'))

labels = pickle.load(open('labels.pkl','rb'))

**To run our model we have to provide the input in the same way as we have done while creating our model. So for this we have created a function which will perform text operations and then predict the label.**

**def** bank\_of\_words(s,words, show\_details=**True**):

bag\_of\_words = [0 **for** \_ **in** range(len(words))]

sent\_words = nltk.word\_tokenize(s)

sent\_words = [lemmatizer.lemmatize(word.lower()) **for** word **in** sent\_words]

**for** sent **in** sent\_words:

**for** i,w **in** enumerate(words):

**if** w == sent:

bag\_of\_words[i] = 1

**return** np.array(bag\_of\_words)

**def** predict\_label(s, model):

# filtering out predictions

pred = bank\_of\_words(s, words,show\_details=**False**)

response = model.predict(np.array([pred]))[0]

ERROR\_THRESHOLD = 0.25

final\_results = [[i,r] **for** i,r **in** enumerate(response) **if** r>ERROR\_THRESHOLD]

final\_results.sort(key=lambda x: x[1], reverse=**True**)

return\_list = []

**for** r **in** final\_results:

return\_list.append({"intent": labels[r[0]], "probability": str(r[1])})

**return** return\_list

**After prediction, now we will create a function which will give responses from the list of intents.**

**def** Response(ints, intents\_json):

tags = ints[0]['intent']

list\_of\_intents = intents\_json['intents']

**for** i **in** list\_of\_intents:

**if**(i['tag']== tags):

response = random.choice(i['responses'])

break

**return** response

**def** chatbot\_response(msg):

ints = predict\_label(msg, model)

response = Response(ints, intents)

**return** response

**Now after responses in this we have created a function which will make user and Bot interact:**

**def** chat():

print("Start chat with ChatBot of ProjectGurukul")

**while** **True**:

inp = input("You: ")

**if** inp.lower() == 'quit':

break

response = chatbot\_response(inp)

print("\n BOT: " + response + '\n\n')

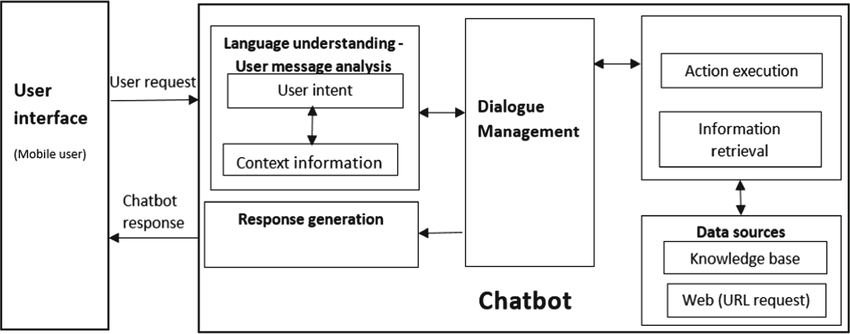
chat()

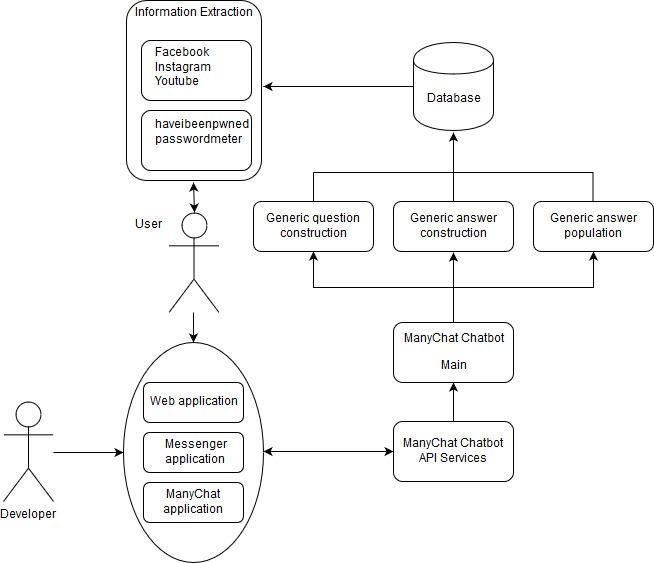
Design, architecture and code implementation:

Design:

There are two approaches that can be used to develop a chatbot depending on the algorithms and techniques adopted: rule-based approach and machine learning approach. Rule-based A rule-based chatbot processes information and provides responses based on a set of predefined rules with the use of pattern matching algorithms. Although the pattern matching techniques vary in complexity, the basic idea is the same. The user input is classified as a pattern, and the chatbot selects a predefined answer by matching the pattern with a set of stored responses. The pattern and response matching algorithms are handcrafted [65]. Pattern matching is adopted by many chatbots and is especially popular among the early chatbots like ELIZA, PARRY, and ALICE. The advantage of the rule-based approach is its speed as it does not require any deep analysis of the input text . However, the responses are repeated and lack flexibility and originality as the knowledge is set by the developer in advance [78]. The following paragraphs will provide an overview of the three most commonly used languages for the implementation of rule-based chatbots. Artificial Intelligence Mark-up Language (AIML) is a derivation of Extensible Mark-up Language (XML) . ALICE was the first chatbot implemented in the AIML language . AIML has a class of data object called an AIML objects, and these objects are responsible for modeling conversational pattern. Each object consists of two units called topics and categories. A topic is an optional top-level element that has a name attribute and a set of categories associated with it. Categories are the most basic unit of knowledge and are the rules of the chatbot. Each category consists of two elements called pattern and template. The pattern matches against the input from the user to the template that contains the response of the chatbot . AIML is simple, flexible, and highly maintainable, and thus is one of the most commonly adopted languages for chatbot development . The biggest disadvantages of AIML are that the developer must specify a pattern for every possible input of the user and that it is inefficient when the knowledge base is large . The structure of an AIML object is demonstrated as below: User Input Corresponding Response to Input . RiveScript is a line-based scripting language that can be used to implement the Knowledge Base . Compared to AIML, RiveScript has more built-in features and tags, which means that the writer does not need to specify information about the chatbot in the additional configuration files. ChatScript is an open-source language for developing rule-based chatbots. It matches user inputs to outputs using rules created by human writers in program scripts through a process called dialog flow scripting . ChatScript uses concepts that are set of words that have the same meaning. 8 It consists of 2000 predefined concepts and scripters can also write their own concepts easily . Compared to AIML and RiveScript, ChatScript is a harder language, but it allows developers to combine rules in more complicated ways.

Architecture:





Implementation details:

The project consists of several parts, the most important of which are: 1- Actions section, which is the folder that contains 3 files in the Python programming language.The most important file is the Actions file, which contains custom Actions that are built according to the need and purpose of the chatbot. 2- The data section and contains 3 important files that cannot be dispensed with: the natural language understanding file, which contains the training data necessary for the bot, which it is expected to receive during its operation from the user, and the rules file, which contains a certain structure that makes the bot act obligatory according to what exists, regardless of the circumstances in terms of the received data, and the story file, which contains scenarios of conversations with users, and all conversations are recorded within this file in .yml extension. 3- There is also a very important file, which is the domain that defines the universe in which your assistant operates. It specifies the intents, entities, slots, responses, forms, and actions your bot should know about. It also defines a configuration for conversation sessions. 4- There are also configuration files, endpoints, and credentials that are responsible for the overall properties and some permissions allowed for the bot and linking with chat channels such as Slack and Facebook Messenger. 5- There is also the Models section, in which all models are stored after each bot training. Every model we can say is like the nucleus or brain of the bot. The bot cannot work and listen to the user’s messages and respond to them without the model. After each bot training process, and to get the latest results, you must choose the newest model. Older models can be selected so that they can be compared with the new model in terms of additions.

Code implementation:

# 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, "-----")

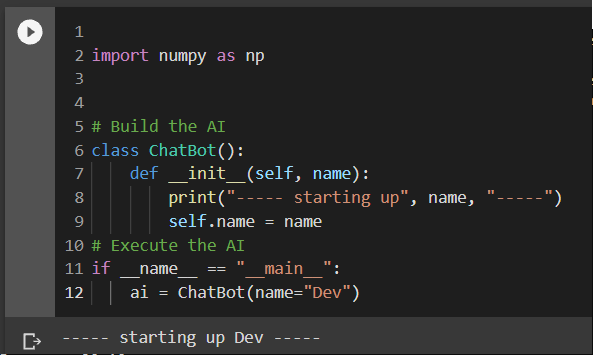
self.name = name

# Execute the AI

**if** \_\_name\_\_ == "\_\_main\_\_":

ai = **ChatBot**(name="Dev")

**Output :**

 **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

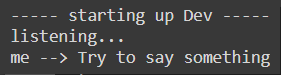
**if** \_\_name\_\_ == "\_\_main\_\_":

ai = ChatBot(name="Dev")

**while** True:

ai.speech\_to\_text()

**Output :**



**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()

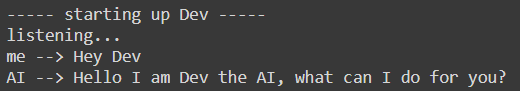
## wake up

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 :**



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(

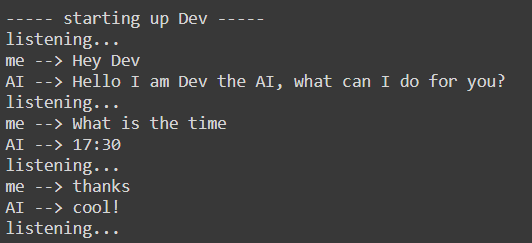
["you're welcome!","anytime!",

"no problem!","cool!",

"I'm here if you need me!","peace out!"])

ai.text\_to\_speech(res)

**Output :**



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](https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)) for our chatbot. This model was presented by Google and it replaced the earlier traditional sequence to sequence models with [attention mechanisms](https://en.wikipedia.org/wiki/Attention_(machine_learning)). 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](https://en.wikipedia.org/wiki/BERT_(language_model)) and OpenAI’s [GPT](https://en.wikipedia.org/wiki/GPT-3). These models have multidisciplinary functionalities and billions of parameters which helps to improve the chatbot and make it truly intelligent.

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](https://huggingface.co/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](https://huggingface.co/transformers/main_classes/pipelines.html) tasks. In the code below, we have specifically used the [DialogGPT](https://huggingface.co/transformers/model_doc/dialogpt.html" \t "_blank) 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.

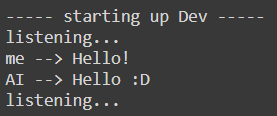
**Code:**

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:



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)

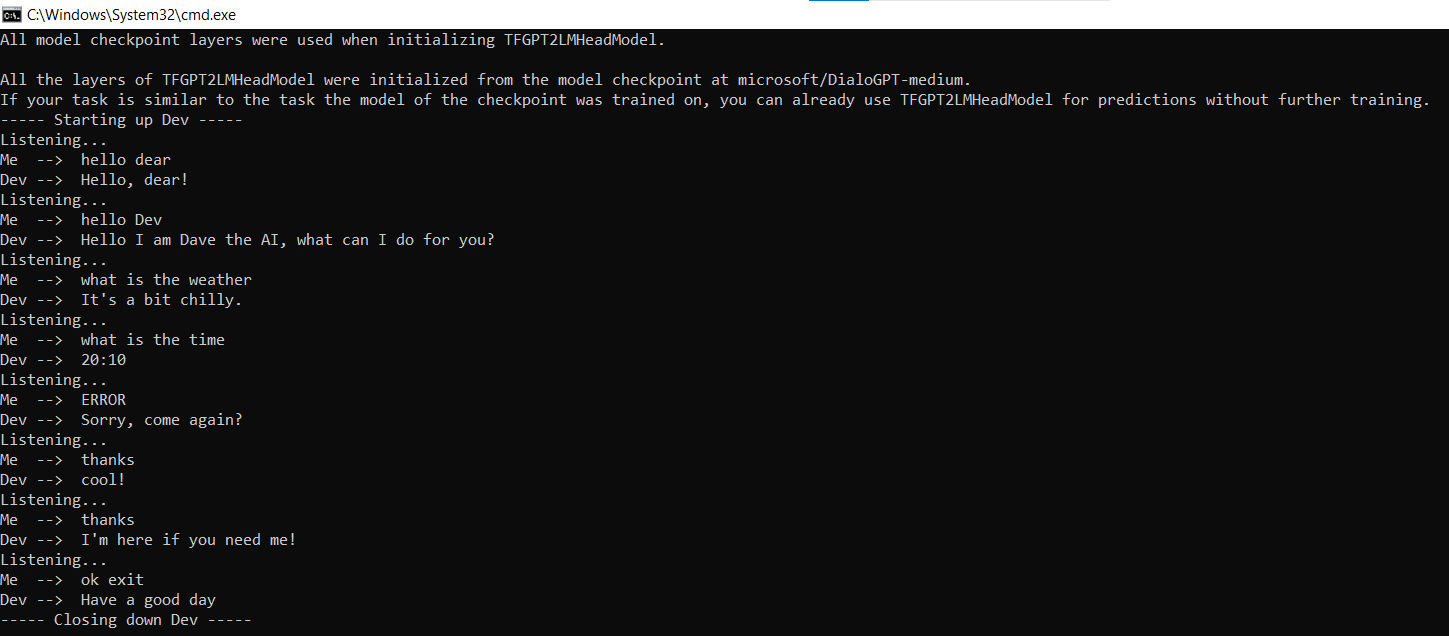
res = str(chat)

res = res[res.find("bot >> ")+6:].strip()

ai.text\_to\_speech(res)

print("----- Closing down Dev -----")

**Output:**



**Note:**I had later switched from google collab to my local machine due to some module issues which I faced during implementation and hence I am sharing my experience here so that if any of you also face the same issue can solve it. Obviously, Google is also there but the following lines will explain the issue. I used Python 3.9 as it had all the modules necessary and Python 3.6 and older versions will also work. Python 3.8 or the latest version might not have all the modules ported to match the version and hence I would suggest using Python 3.9 or older versions than 3.6.

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.

The link to the full code can be found [here](https://github.com/arnabm14/Dev_AIChatbot_NLP).

**Bonus tips:** Feel free to drop a star if you liked this tutorial or bot and feel free to fork and create your own AI chatbot and call it whatever you want!

**Conclusion**

In this guide, we’ve provided a step-by-step tutorial for creating a conversational chatbot. You can use this chatbot as a foundation for developing one that communicates like a human.