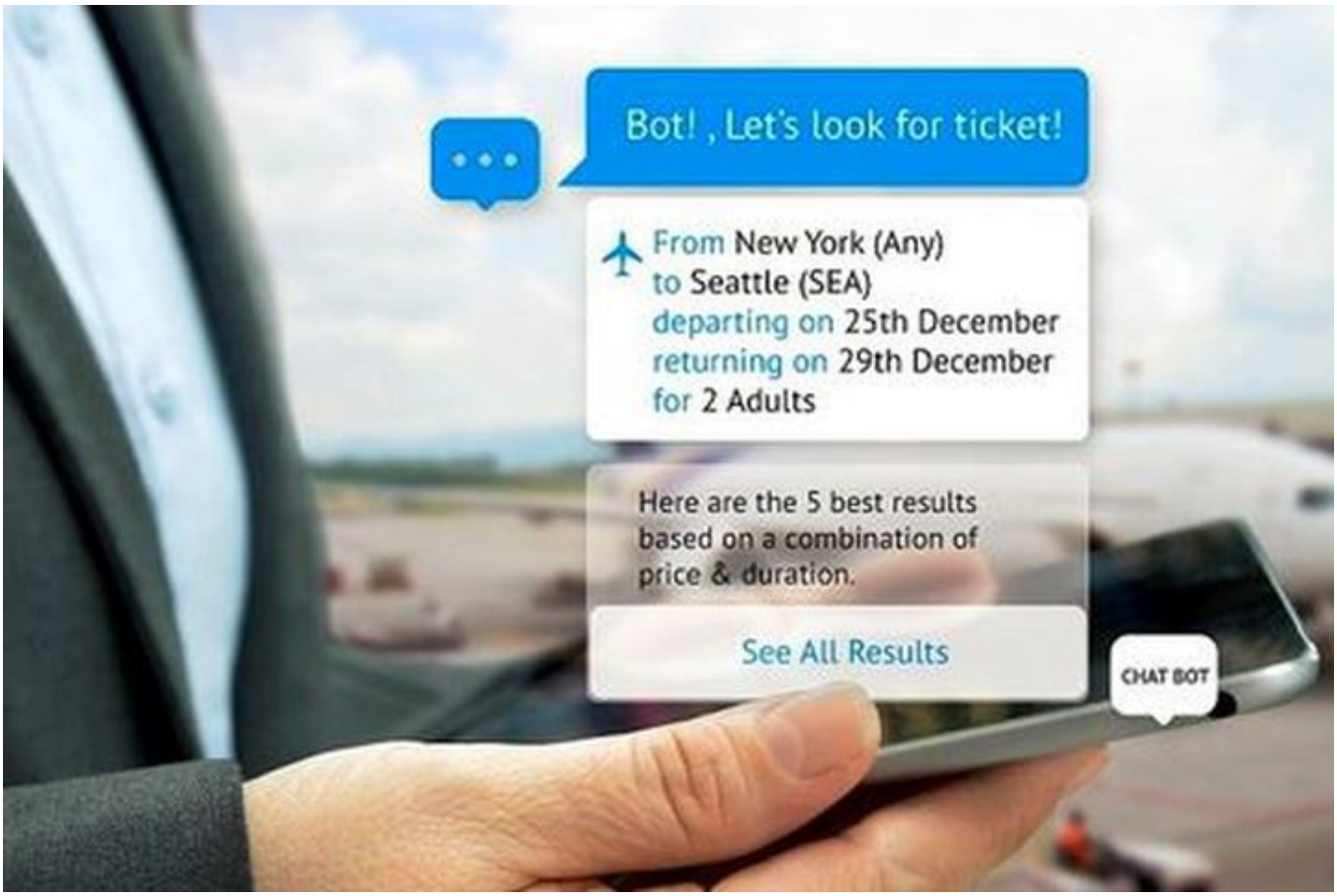
Building a Simple Chatbot from Scratch in Python (using NLTK)

Parul Pandey
Sep 17, 2018 · 11 min read



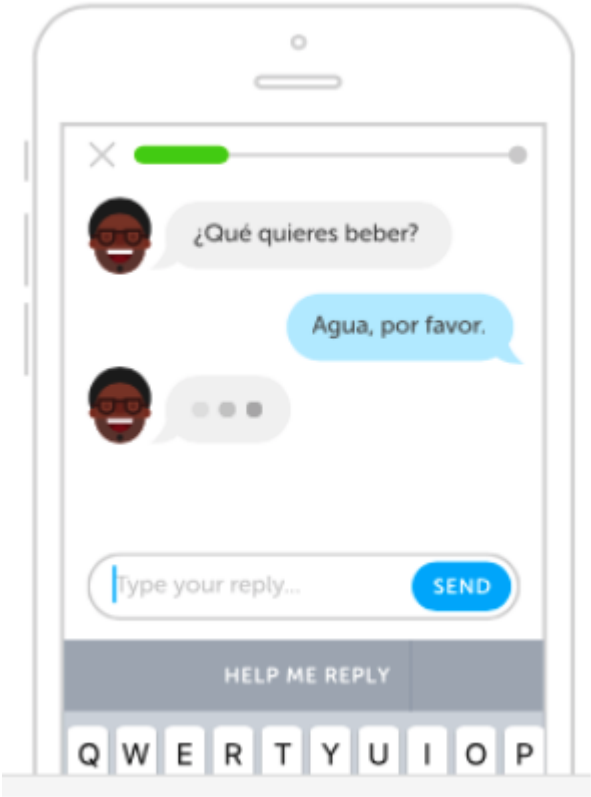
Source: eWeek

> Gartner estimates that by 2020, chatbots will be handling 85 percent of customer-service interactions; they are already handling about 30 percent of transactions now.

I am sure you've heard about ***Duolingo****: a popular language-learning app, which gamifies practicing a new language. It is quite popular due to its innovative styles of teaching a foreign language.The concept is simple: five to ten minutes of interactive training a day is enough to learn a language.*

However, even though Duolingo is enabling people to learn a new language, it's practitioners had a concern. People felt they were missing out on learning valuable conversational skills since they were learning on their own. People were also apprehensive about being paired with other language learners due to fear of embarrassment. This was turning out be a big bottleneck in Duolingo's plans.

So their team solved this problem by building a native chatbot within its app, to help users learn conversational skills and practice what they learned.



http://bots.duolingo.com/

Since the bots are designed as conversational and friendly, Duolingo learners can practice conversation any time of the day, using their choice of characters, until they feel brave enough to practice their new language with other speakers. ***This solved a major consumer pain point and made learning through the app a lot more fun.***

. . .

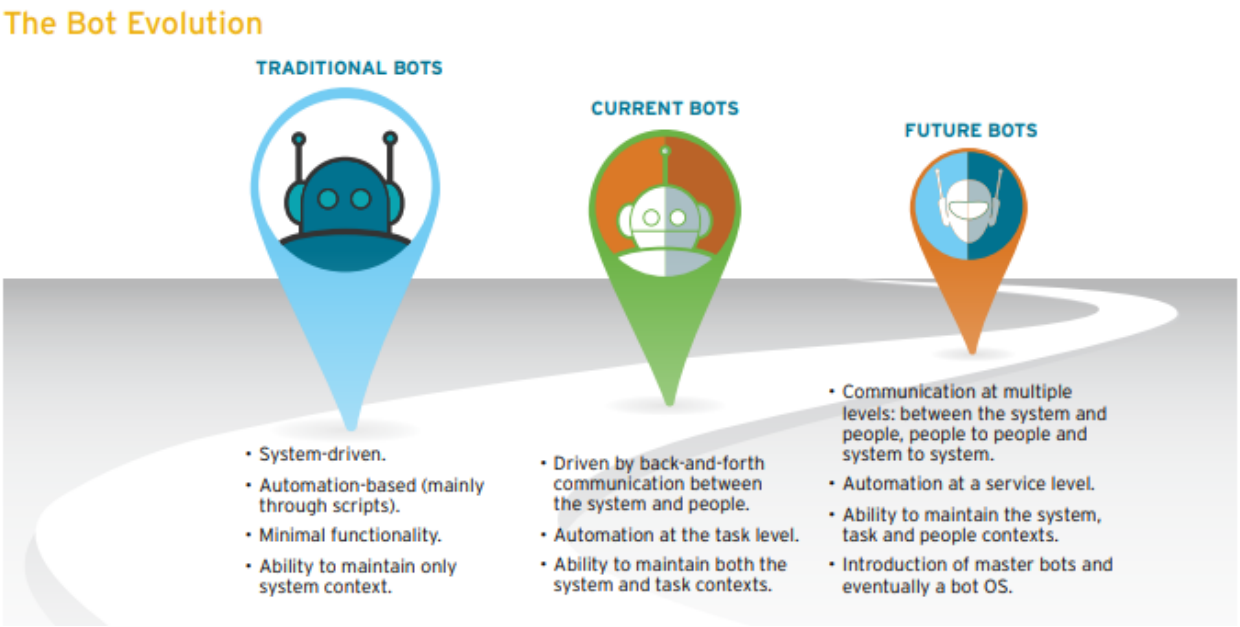
So what is a chatbot?

A **chatbot** is an artificial intelligence-powered piece of software in a device (Siri, Alexa, Google Assistant etc), application, website or other networks that try to gauge consumer's needs and then assist them to perform a particular task like a commercial transaction, hotel booking, form submission etc . Today almost every company has a chatbot deployed to engage with the users. Some of the ways in which companies are using chatbots are:

- To deliver flight information
- to connect customers and their finances
- As customer support

The possibilities are (almost) limitless.

> ***History of chatbots dates back to 1966 when a computer program called ELIZA was invented by Weizenbaum. It imitated the language of a psychotherapist from only 200 lines of code. You can still converse with it here: Eliza.***



Source: Cognizant

How do Chatbots work?

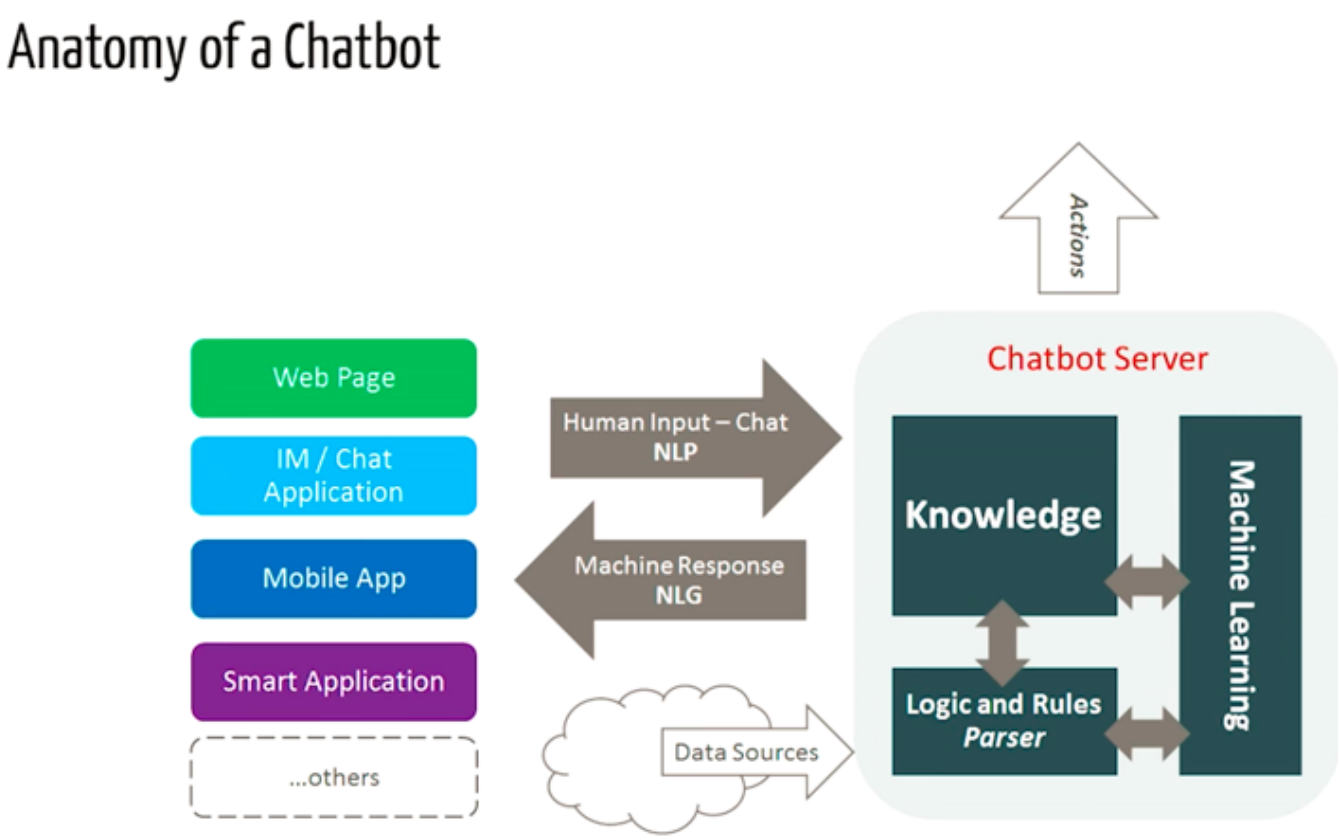
There are broadly two variants of chatbots: **Rule-Based** and **Self-learning.**

1. 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 bots can handle simple queries but fail to manage complex ones.

2. Self-learning bots are the ones that use some Machine Learning-based approaches and are definitely more efficient than rule-based bots. These bots can be of further two types: **Retrieval Based** or **Generative**

i) In **retrieval-based models**, a chatbot uses some heuristic to select a response from a library of predefined responses. The chatbot uses the message and context of the conversation for selecting the best response from a predefined list of bot messages. The context can include a current position in the dialogue tree, all previous messages in the conversation, previously saved variables (e.g. username). Heuristics for selecting a response can be engineered in many different ways, from rule-based if-else conditional logic to machine learning classifiers.

ii) **Generative** bots can generate the answers and not always replies with one of the answers from a set of answers. This makes them more intelligent as they take word by word from the query and generates the answers.



> *In this article we will build a simple retrieval based chatbot based on NLTK library in python.*

Building the Bot

Pre-requisites

Hands-On knowledge of **scikit** library and **NLTK** is assumed. However, if you are new to NLP, you can still read the article and then refer back to resources.

NLP

The field of study that focuses on the interactions between human language and computers is called Natural Language Processing, or NLP for short. It sits at the intersection of computer science, artificial intelligence, and computational linguistics[Wikipedia].NLP is a way for computers to analyze, understand, and derive meaning from human language in a smart and useful way. By utilizing NLP, developers can organize and structure knowledge to perform tasks such as automatic summarization, translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation.

NLTK: A Brief Intro

NLTK(Natural Language Toolkit) is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries.

NLTK has been called “a wonderful tool for teaching and working in, computational linguistics using Python,” and “an amazing library to play with natural language.”

Natural Language Processing with Python provides a practical introduction to programming for language processing. I highly recommend this book to people beginning in NLP with Python.

Downloading and installing NLTK

1. Install NLTK: run `pip install nltk`
2. Test installation: run `python` then type `import nltk`

For platform-specific instructions, read here.

Installing NLTK Packages

import NLTK and run `nltk.download()`. This will open the NLTK downloader from where you can choose the corpora and models to download. You can also download all packages at once.

Text Pre- Processing with NLTK

The main issue with text data is that it is all in text format (strings). However, Machine learning algorithms need some sort of numerical feature vector in order to perform the task. So before we start with any NLP project we need to pre-process it to make it ideal for work. Basic **text pre-processing** includes:

- Converting the entire text into **uppercase or lowercase**, so that the algorithm does not treat the same words in different cases as different
- **Tokenization**: Tokenization is just the term used to describe the process of converting the normal text strings into a list of tokens i.e words that we actually want. Sentence tokenizer can be used to find the list of sentences and Word tokenizer can be used to find the list of words in strings.

The NLTK data package includes a pre-trained Punkt tokenizer for English.

- Removing **Noise** i.e everything that isn't in a standard number or letter.
- Removing **Stop words.** Sometimes, some extremely common words which would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely. These words are called *stop words*
- **Stemming**: Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base or root form — generally a written word form. Example if we were to stem the following words: "Stems", "Stemming", "Stemmed", "and Stemtization", the result would be a single word "stem".
- **Lemmatization**: A slight variant of stemming is lemmatization. The major difference between these is, that, stemming can often create non-existent words, whereas lemmas are actual words. So, your root stem, meaning the word you end up with, is not something you can just look up in a dictionary, but you can look up a lemma. Examples of Lemmatization are that "run" is a base form for words like "running" or "ran" or that the word "better" and "good" are in the same lemma so they are considered the same.

Bag of Words

After the initial preprocessing phase, we need to transform the text into a meaningful vector (or array) of numbers. The bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.

- A measure of the presence of known words.

Why is it is called a "*bag*" of words? That is because any information about the order or structure of words in the document is discarded and the model is only concerned with **whether the known words occur in the document, not where they occur in the document.**

The intuition behind the Bag of Words is that documents are similar if they have similar content. Also, we can learn something about the meaning of the document from its content alone.

For example, if our dictionary contains the words {Learning, is, the, not, great}, and we want to vectorize the text "Learning is great", we would have the following vector: (1, 1, 0, 0, 1).

TF-IDF Approach

A problem with the Bag of Words approach is that highly frequent words start to dominate in the document (e.g. larger score), but may not contain as much "informational content". Also, it will give more weight to longer documents than shorter documents.

One approach is to rescale the frequency of words by how often they appear in all documents so that the scores for frequent words like "the" that are also frequent across all documents are penalized. This approach to scoring is called **Term Frequency-Inverse Document Frequency**, or **TF-IDF** for short, where:

Term Frequency: is a scoring of the frequency of the word in the current document.

```
TF = (Number of times term t appears in a document)/(Number of terms
in the document)
```

Inverse Document Frequency: is a scoring of how rare the word is across documents.

```
IDF = 1+log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.
```

Tf-IDF weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus

Example:

> *Consider a document containing 100 words wherein the word 'phone' appears 5 times.*
>
> *The term frequency (i.e., tf) for phone is then (5 / 100) = 0.05. Now, assume we have 10 million documents and the word phone appears in one thousand of these. Then, the inverse document frequency (i.e., IDF) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-IDF weight is the product of these quantities: 0.05 * 4 = 0.20.*
>
> *Tf-IDF can be implemented in scikit learn as:*
>
> *from sklearn.feature_extraction.text import TfidfVectorizer*

Cosine Similarity

TF-IDF is a transformation applied to texts to get two real-valued vectors in vector space. We can then obtain the **Cosine** similarity of any pair of vectors by taking their dot product and dividing that by the product of their norms. That yields the cosine of the angle between the vectors. **Cosine similarity** is a measure of similarity between two non-zero vectors. Using this formula we can find out the similarity between any two documents d1 and d2.

```
Cosine Similarity (d1, d2) =  Dot product(d1, d2) / ||d1|| * ||d2||
```

where d1,d2 are two non zero vectors.

For a detailed explanation and practical example of TF-IDF and Cosine Similarity refer to the document below.

Tf-Idf and Cosine similarity

In the year 1998 Google handled 9800 average search queries every day. In

2012 this number shot up to 5.13 billion...

janav.wordpress.com

. . .

Now we have a fair idea of the NLP process. It is time that we get to our real task i.e Chatbot creation. We will name the chatbot here as 'ROBO🤖'.

> You can *find the entire code with the corpus at the associated Github Repository here or you can view it on my binder by clicking the image below.*

Importing the necessary libraries

```
import nltk
import numpy as np
import random
import string # to process standard python strings
```

Corpus

For our example, we will be using the Wikipedia page for chatbots as our corpus. Copy the contents from the page and place it in a text file named 'chatbot.txt'. However, you can use any corpus of your choice.

Reading in the data

We will read in the corpus.txt file and convert the entire corpus into a list of sentences and a list of words for further pre-processing.

```
f=open('chatbot.txt','r',errors = 'ignore')
```

```
raw=f.read()

raw=raw.lower()# converts to lowercase

nltk.download('punkt') # first-time use only
nltk.download('wordnet') # first-time use only

sent_tokens = nltk.sent_tokenize(raw)# converts to list of sentences
word_tokens = nltk.word_tokenize(raw)# converts to list of words
```

Let see an example of the sent_tokens and the word_tokens

```
sent_tokens[:2]
['a chatbot (also known as a talkbot, chatterbot, bot, im bot,
interactive agent, or artificial conversational entity) is a
computer program or an artificial intelligence which conducts a
conversation via auditory or textual methods.',
 'such programs are often designed to convincingly simulate how a
human would behave as a conversational partner, thereby passing the
turing test.']

word_tokens[:2]
['a', 'chatbot', '(', 'also', 'known']
```

Pre-processing the raw text

We shall now define a function called LemTokens which will take as input the tokens and return normalized tokens.

```
lemmer = nltk.stem.WordNetLemmatizer()
#WordNet is a semantically-oriented dictionary of English included
in NLTK.

def LemTokens(tokens):
    return [lemmer.lemmatize(token) for token in tokens]
remove_punct_dict = dict((ord(punct), None) for punct in
string.punctuation)
def LemNormalize(text):
    return
LemTokens(nltk.word_tokenize(text.lower().translate(remove_punct_dic
t)))
```

Keyword matching

Next, we shall define a function for a greeting by the bot i.e if a user's input is a greeting, the bot shall return a greeting response.ELIZA uses a simple keyword

matching for greetings. We will utilize the same concept here.

```
GREETING_INPUTS = ("hello", "hi", "greetings", "sup", "what's
up","hey",)

GREETING_RESPONSES = ["hi", "hey", "*nods*", "hi there", "hello", "I
am glad! You are talking to me"]

def greeting(sentence):

    for word in sentence.split():
        if word.lower() in GREETING_INPUTS:
            return random.choice(GREETING_RESPONSES)
```

Generating Response

To generate a response from our bot for input questions, the concept of document similarity will be used. So we begin by importing the necessary modules.

- From scikit learn library, import the TFidf vectorizer to convert a collection of raw documents to a matrix of TF-IDF features.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

- Also, import cosine similarity module from scikit learn library

```
from sklearn.metrics.pairwise import cosine_similarity
```

This will be used to find the similarity between words entered by the user and the words in the corpus. This is the simplest possible implementation of a chatbot.

We define a function **response** which searches the user's utterance for one or more known keywords and returns one of several possible responses. If it doesn't find the input matching any of the keywords, it returns a response:" I am sorry! I don't understand you"

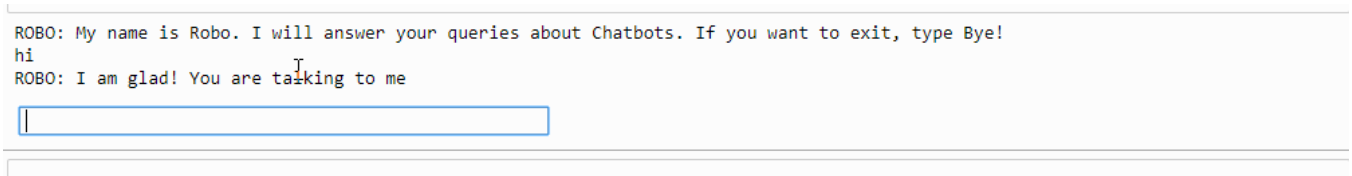
```
def response(user_response):
    robo_response=''
    sent_tokens.append(user_response)
```

```
    TfidfVec = TfidfVectorizer(tokenizer=LemNormalize,
stop_words='english')
    tfidf = TfidfVec.fit_transform(sent_tokens)
    vals = cosine_similarity(tfidf[-1], tfidf)
    idx=vals.argsort()[0][-2]
    flat = vals.flatten()
    flat.sort()
    req_tfidf = flat[-2]

    if(req_tfidf==0):
        robo_response=robo_response+"I am sorry! I don't understand
you"
        return robo_response
    else:
        robo_response = robo_response+sent_tokens[idx]
        return robo_response
```

Finally, we will feed the lines that we want our bot to say while starting and ending a conversation depending upon the user's input.

```
flag=True
print("ROBO: My name is Robo. I will answer your queries about
Chatbots. If you want to exit, type Bye!")

while(flag==True):
    user_response = input()
    user_response=user_response.lower()
    if(user_response!='bye'):
        if(user_response=='thanks' or user_response=='thank you' ):
            flag=False
            print("ROBO: You are welcome..")
        else:
            if(greeting(user_response)!=None):
                print("ROBO: "+greeting(user_response))
            else:
                print("ROBO: ",end="")
                print(response(user_response))
                sent_tokens.remove(user_response)
    else:
        flag=False
        print("ROBO: Bye! take care..")
```

So that's pretty much it. We have coded our first chatbot in NLTK. Now, let us see how it interacts with humans:



This wasn't too bad. Even though the chatbot couldn't give a satisfactory answer for some questions, it fared pretty well on others.

Conclusion

Though it is a very simple bot with hardly any cognitive skills, its a good way to get into NLP and get to know about chatbots.Though 'ROBO' responds to user input. It won't fool your friends, and for a production system you'll want to consider one of the existing bot platforms or frameworks, but this example should help you think through the design and challenge of creating a chatbot. Internet is flooded with resources and after reading this article I am sure , you will want to create a chatbot of your own. So happy tinkering!!

Data Science Artificial Intelligence Chatbots NLP Machine Learning

About Help Legal