MMLProject

Nitasha Fazal

April 2024

1 Project: Chatbot

1.1 Introduction

In the contemporary digital landscape, the term "chatbot" has become increasingly prevalent. Before diving into the intricacies of its implementation and development, it's essential to establish a foundational understanding of what chatbots are, where they are utilized, what functions they perform, and why they have become indispensable in various domains.

What are Chatbots?

Chatbots are computer programs designed to simulate conversation with human users, typically through text or voice interfaces. They leverage artificial intelligence (AI), natural language processing (NLP), and machine learning (ML) algorithms to understand user queries, discern the intent, and context and provide relevant responses or actions.

Where are they Used?

Chatbots are used for various tasks, from customer support to information retrieval, providing instant communication in e-commerce, healthcare, banking, education, and other diverse applications. We can see chat bots on almost all websites these days.

What Do They Do?

Chat bots like all other software application intent to fascilitate humans and reduce human efforts and interventions. They streamline processes, and enhance user experience, like Customer Support, where they provide instant assistance, answer queries, and resolve issues for customers round-the-clock.

Why Do We Need Them?

Chatbots enhance user experience, and are best for cost saving. Users / customers can get instant, personalized, and convenient interactions without any expert human resource.

1.2 Various Implementations of Chatbot

There are various implementation of chatbots from simple to complex, depending on the need, we can develop one.

Simple rule based chatbot engine is easy to implement but lacks flexibility. Rules are being created by the developers, and chatbot can only answer the questions for given questions in a predefined format. These chatbots lacks the ability to cater any other query.
Classifier In the realm of supervised learning, a classification chatbot engine, commonly known as a bots, utilizes its machine learning capabilities to discern context and assign labels based on patterns or words found within the query. And then they respond according to that label. These labels have predefined responses already stored in database.
Predictor These chatbot engines operate on unsupervised machine learning models, where unlabeled data is inputted into the system. Through this process, the model autonomously learns to predict and respond by discerning sequences and patterns within the input data. This enables the chatbot to generate relevant responses without explicit labeling or guidance, leveraging the inherent structure and correlations present in the data to facilitate natural and contextually appropriate conversations.

1.3 Project Implementation Overview

In this project, I've developed a chatbot powered by a classifier engine, employing supervised learning techniques. The dataset I utilized was sourced from three distinct repositories, outlined in the subsequent section. This dataset comprises labeled entries featuring three distinct tags, along with corresponding questions/patterns and responses. Leveraging a neural network, the chatbot learns the semantic nuances of the input patterns, enabling it to classify the appropriate labels from the given set and generate corresponding responses.

1.4 Data

The data I used was in json files with the following structure.

```
{"intents": [
   {
    "tag": "hello",
    "patterns": [ "Hello","Hi there","Good morning",
                  "What's up ],
    "responses": ["Hey!","Hello","Hi!","Good morning!" ],
    "context": ""
  },
  { "tag": "noanswer",
    "patterns": [],
    "responses": ["Sorry, can't understand you",
    "Please give me more info", "Not sure I understand" ],
    "context": [""]
  },
```

```
  {
    "tag": "job",
    "patterns": ["What is your job", "What is your work"],
    "responses":["My job is to make you
                  feel like everything is okay.",
      "I work to serve you as well as possible"],
    "context": ""
  }
]}
```

This intend file has data that is being gathered from 3 different resources references mentioned in the references. Here we have 4 tags per data entry, labels, patterns and responses, I have not used the context, but we can train the model according to the context also, to improve its accuracy. This will make model more accurate as we will classify not only on words but also with the context of the querie.

1.5 Data Processing

Now that we've familiarized ourselves with the data structure, let me outline the process of preparing it for model ingestion. Our dataset is comprised of natural language, specifically English. However, machine learning models operate on numerical data, necessitating a preprocessing step to transform our text-based data into a suitable format

Pattern Tokenizing In the data preparation phase, the initial step involves tokenizing the patterns/queries to obtain the vocabulary of words. TensorFlow, a Python library, along with the Natural Language Toolkit (NLTK), offer this capability. When employing TensorFlow's Tokenizer class, the vocabulary is automatically stored within its parameters for future use. Conversely, when utilizing NLTK, it's necessary to manually store the vocabulary of tokenized words. In my approach, I utilized NLTK's tokenize function for this purpose.

```
print(len(words), "lemmatized words", words)
```

```
696 tokenized words ['Hello', 'Hi', 'there', 'Good', 'morning', 'What', "'s", 'up', 'What', 'is', 'your', 'job', 'What', 'is', 'your', 'work', 'What',
'is', 'your', 'age', 'How', 'old', 'are', 'you', 'When', 'were', 'you', 'born', 'How', 'are', 'you', 'today', 'How', 'are', 'you', 'I', 'am', 'good',
'too', 'I', 'feel', 'fine', 'Good', '!', 'Fine', 'I', 'am', 'good', 'I', 'am', 'great', 'great', 'I', 'am', 'feeling', 'bad', 'No', 'I', 'am', 'sad',
'No', 'What', 'can', 'you', 'do', 'What', 'can', 'I', 'ask', 'you', 'Can', 'you', 'help', 'me', 'Are', 'you', 'a', 'girl', 'You', 'are', 'a', 'women',
'Are', 'you', 'a', 'men', 'Are', 'you', 'a', 'boy', 'Thank', 'you', 'Thank', 'you', 'very', 'much', 'thanks', 'Goodbye', 'Good', 'afternoon', 'Bye', 'W
here', 'do', 'you', 'live', 'What', 'are', 'you', 'doing', 'Can', 'you', 'wait', '2', 'minutes', 'Please', 'wait', 'Wait', '2', 'secs', 'please', 'Ar
e', 'you', 'still', 'there', '?', 'Are', 'you', 'here', '?', 'hi', 'there', 'hello', 'haroo', 'yaw', 'wassup', 'hi', 'hey', 'holla', 'hello', 'bye', 'g
ood', 'bye', 'see', 'you', 'later', 'Thanks', 'okay', 'Thank', 'you', 'thankyou', 'That', "'s", 'helpful', 'Awesome', ',', 'thanks', 'Thanks', 'for',
'helping', 'me', 'wow', 'great', 'what', "'s", 'your', 'name', '?', 'who', 'are', 'you', '?', 'my', 'name', 'is', 'I', "'m", 'I', 'am', 'coffee', '?',
'can', 'i', 'take', 'you', 'out', 'on', 'a', 'date', 'I', 'need', 'a', 'favour', 'can', 'you', 'help', 'me', 'I', 'need', 'you', 'All', 'I', 'need', 'i
s', 'you', 'I', 'want', 'you', 'What', 'is', 'AI', '?', 'Are', 'you', 'sentient', '?', 'Are', 'you', 'sapient', '?', 'wtf', 'What', 'language', 'are',
'you', 'written', 'in', '?', 'You', 'sound', 'like', 'Data', 'You', 'are', 'an', 'artificial', 'linguistic', 'entity', 'You', 'are', 'not', 'immortal',
'Are', 'you', 'immortal', '?', 'You', 'are', 'not', 'making', 'sense', 'You', 'can', 'not', 'clone', 'You', 'can', 'not', 'move', 'When', 'will', 'yo
u', 'walk', 'Can', 'you', 'walk', 'Can', 'you', 'move', 'Bend', 'over', 'Can', 'you', 'mate', 'Robots', 'laugh', 'Robots', 'should', 'die', 'When', 'd
o', 'you', 'die', 'I', 'hope', 'that', 'you', 'die', 'I', 'do', 'not', 'want', 'to', 'die', 'Can', 'you', 'die', 'Robots', 'Robots', 'are', 'stupid',
'Are', 'you', 'stupid', 'Robots', 'are', 'not', 'allowed', 'to', 'lie', 'Robotics', 'It', 'is', 'a', 'computer', 'When', 'will', 'you', 'fight', 'Wha
t', 'is', 'a', 'chat', 'robot', '?', 'What', 'is', 'a', 'chatterbox', 'What', 'is', 'a', 'motormouth', 'What', 'is', 'a', 'ratchet', 'jaw', 'What', 'i
s', 'your', 'robot', 'body', 'What', 'is', 'your', 'business', 'What', 'is', 'your', 'favorite', 'programming', 'language', 'What', 'is', 'your', 'favo
rite', 'hobby', 'What', 'do', 'you', 'like', 'to', 'do', '?', 'What', 'is', 'your', 'idea', 'What', 'is', 'your', 'shoe', 'size', 'What', 'is', 'it',
```

Figure 1: Tokenized words vocabulary in project.

Words lemmatization In this particular step, I normalized various forms of

verbs into their base or lemma form. For instance, words like 'builds,' 'building,' or 'built' were all reduced to the lemma 'build.' Once more, I employed the NLTK library to accomplish this task.

Ignore Punctuations

I ignored the punctuation's like question mark, full stop, comma from these patterns. We can also clean our dictionary by cleaning digits and single alphabets also, but I just removed the punctuation's.

After this step, I arranged the data in a hashtable / dictionary, where keys are the labels and vocabulary of words in pattern are the values.

```
hello -> ['hello', 'hi', 'there', 'good', 'morning', 'up', 'what']
job -> ['is', 'job', 'what', 'your', 'is', 'what', 'work', 'your']
age -> ['age', 'is', 'what', 'your', 'are', 'how', 'old', 'you', 'born', 'were', 'when', 'you', 'how', 'old', 'dexter', 'how', 'is', 'old', 'age', 'i
s', 'what', 'your', 'are', 'how', 'old', 'you', 'age']
feeling -> ['are', 'how', 'today', 'you', 'are', 'how', 'you']
good -> ['am', 'good', 'i', 'too', 'feel', 'fine', 'i', 'good', 'fine', 'am', 'good', 'i', 'am', 'great', 'i', 'great']
bad -> ['am', 'bad', 'feeling', 'i', 'am', 'i', 'no', 'sad', 'no']
actions -> ['can', 'do', 'what', 'you', 'ask', 'can', 'i', 'what', 'you', 'can', 'help', 'me', 'you']
women -> ['are', 'girl', 'you', 'are', 'woman', 'you']
men -> ['are', 'men', 'you', 'are', 'boy', 'you']
thanks -> ['thank', 'you', 'much', 'thank', 'very', 'you', 'thanks', 'thanks', 'okay', 'thank', 'you', 'thankyou', 'helpful', 'that', 'awesome', 'thank
s', 'for', 'helping', 'me', 'thanks', 'wow', 'great']
goodbye -> ['goodbye', 'afternoon', 'good', 'bye', 'bye', 'bye', 'good', 'later', 'see', 'you', 'cya', 'later', 'see', 'you', 'goodbye', 'am', 'i', 'le
aving', 'day', 'good', 'have', 'cya', 'later', 'go', 'got', 'i', 'now', 'ta', 'got', 'i', 'now', 'rush', 'ta']
city -> ['do', 'live', 'where', 'you']
action -> ['are', 'doing', 'what', 'you']
wait -> ['2', 'can', 'minute', 'wait', 'you', 'please', 'wait', '2', 'please', 'sec', 'wait']
still there -> ['are', 'still', 'there', 'you', 'are', 'here', 'you']
greetings -> ['hi', 'there', 'hello', 'haroo', 'yaw', 'wassup', 'hi', 'hey', 'holla', 'hello']
noanswer -> []
name1 -> ['name', 'what', 'your', 'are', 'who', 'you']
name -> ['is', 'my', 'name', 'i', 'am', 'i', 'is', 'name', 'what', 'your', 'call', 'i', 'should', 'what', 'you', 'name', 'whats', 'your']
date -> ['coffee', 'can', 'date', 'i', 'on', 'out', 'take', 'you']
fav -> ['favour', 'i', 'need', 'can', 'help', 'me', 'you']
need -> ['i', 'need', 'you', 'all', 'i', 'is', 'need', 'you', 'i', 'want', 'you']
AI -> ['ai', 'is', 'what']
sentiment -> ['are', 'sentient', 'you']
sapient -> ['are', 'sapient', 'you']
abbr -> ['wtf']
lang -> ['are', 'in', 'language', 'what', 'written', 'you']
sound -> ['data', 'like', 'sound', 'you']
artificial -> ['an', 'are', 'artificial', 'entity', 'linguistic', 'you']
imortal -> ['are', 'immortal', 'not', 'you', 'are', 'immortal', 'you']
sense -> ['are', 'making', 'not', 'sense', 'you']
clone -> ['can', 'clone', 'not', 'you']
```

Figure 2: Dictionary of tokenized words with labels.

Words to Numeric Once we've structured our data accordingly, we're faced with two choices: employing word embedding and padding, or opting for one-hot encoding to transform our natural language data into numerical form. TensorFlow conveniently offers built-in functionality for embedding. Embedding involves utilizing word indexes from a vocabulary; these indexes correspond to the word sequences within sentences.

Words in vocab: 'a', 'word', 'new', 'Artificial', 'is', 'Intelligence'

Index in vocab: 0, 1, 2, 3, 4, 5

Embedding will be: 0, 2, 1, 4, 3, 5

One-hot encoding typically transforms categorical data into numeric form by employing a binary representation of categories. However, when applied to words, it involves utilizing the entire dataset of words and assigning a value of 1 to the index where a word exists in the vocabulary. Lets say we have 500 words in vocabulary. Now we will create a dataset of values, and place 0 for all words. We will only have 1 for the words of that are present in that particular sentence.

Data Set with size of 500: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.....0,0,0]

One Hot Encoding: [1,1,1,1,1, 0,0,0,0,0,0,0,.....................0,0,0]
Given that the words in this sentence are indexed from 0 to 5, only these 5 indexes have a value of 1, while all other indexes contain 0s. Consequently, there's no necessity for padding in this scenario, as the size of the numeric data for all sentences remains consistent.
Padding The length of each query sentence may vary, but our machine learning models typically operate on numeric inputs of consistent length. To achieve this, we pad our input queries with additional 0s at either the beginning or the end. This aids in the learning process of the neural networks. If we opt one hot encoding, padding is already there. but if we opt embedding, we usually need to pad the embedded numeric sentences with 0's.
Embedding sentence was : 0, 2, 1, 4, 3, 5
but lets say in our data, the largest sentence length is 25 words, now we will embed all other sentences before feeding it into neural network with 0's to make all input's data length consistent.
After padding the sentence will be: 0,2,1,4,3,5,0,0,0,0,0,0,0,0,0,0,0,0......0,0
In my project I have opted the one hot encoding. Two rows from the training data are as follows:

```
Training dataset size:  196
Row 1
Bag of Words: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Output Row: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Row 2
Bag of Words: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Output Row: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 3: One hot encoding for neural network input feed

1.6 Neural Network as classification Model

Now that our data is prepared, it's time to choose a model for training. We need to train a model so that it can classify the labels for new queries. While there are options like random forest or decision trees which can be used, I've decided to go with a neural network. This will allow us to train a model that can effectively classify labels on unknown data with greater flexibility and potentially higher accuracy. Now in the realm of neural network, we have couple of options for instance, there's the Sequence-to-sequence (seq2seq) model, often adorned with embedding layers, ideal for tasks like language translation where input and output sequences vary in length. Then, we have Recurrent Neural Networks (RNNs), a stalwart in natural language processing and speech recognition. Moreover, there are specialized architectures like Long Short-Term

Memory networks (LSTMs) and Gated Recurrent Units (GRUs), crafted to mitigate issues like the vanishing gradient problem in traditional RNNs.
These neural network models are particularly potent in unsupervised learning scenarios, where raw data devoid of labels is the norm. By leveraging the sequential nature of the data, these models can learn intricate patterns and structures, enabling them to generate coherent predictions or sentences based solely on the training data sequence.
The scope of our project is limited to supervised learning, where we have data with labels, hence I used rather a simple neural network with hidden layer for classification.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 128)               26624

 dropout_2 (Dropout)         (None, 128)               0

 dense_4 (Dense)             (None, 64)                8256

 dropout_3 (Dropout)         (None, 64)                0

 dense_5 (Dense)             (None, 75)                4875

=================================================================
Total params: 39755 (155.29 KB)
Trainable params: 39755 (155.29 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4: Keras Sequential neural network summary

Accuracy
In order to test the model, we needed to test the learning of model by giving it similar words in query and check if it is able to predict the labels correctly. In order to do this, I created a new json file with 35 classes, and modified patterns with similar words and sentences. Below is the result with confusion matrix.

1.7 Math Behind the Model

The model I used has dense layers, these are fully connected hidden layers of neurons.In a dense layer, each neuron is connected to every neuron in the previous layer
Input layer is the training dataset we prepared in the last section. Length of training dataset is the size of input layer. Numeric representation of the words from question which we prepared in the last section, are being used as features in the neural network. First hidden layer consist of 128 neurons, and the activation function used is "relu".
Second hidden layer is also dense, fully connected layer with same number of

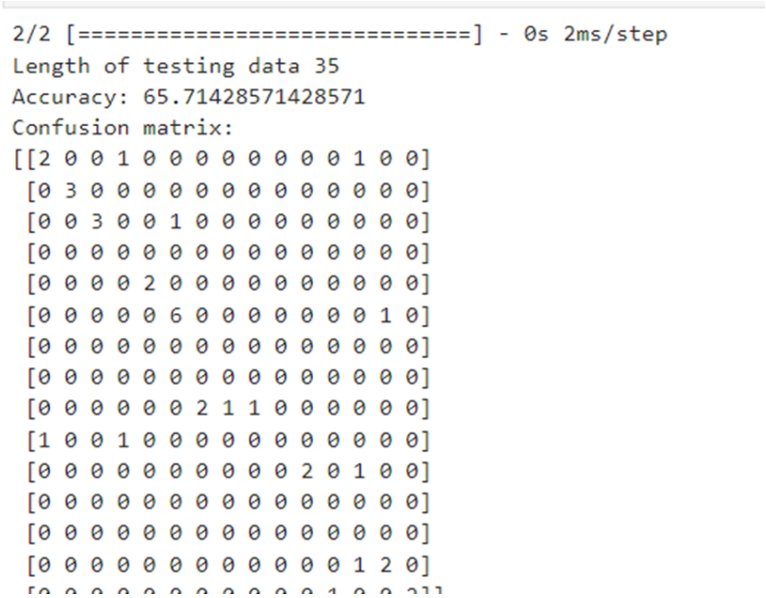

Figure 5: Accuracy of Neural Network Model

neurons. Our third layer is the output layer with activation function "Softmax" because this time we need probabilities for all classes, and will select the class with highest probability.
The output is calculated as follows:
Input to the first layer is x, which is the data / feature set.
multiplying it with weights and adding bias, our z becomes:
$z^{[1]} = W^{[1]}_{n\times m} x_{m\times n} + b^{[1]}_{n\times 1}$
where m = rows of data,
n = columns of features
b = bias for that layer
w = are the weights of each neurons in the layer
for weights, rows are the weights, and columns are the neurons / layer
bias has bias values in rows for each neuron/layer and it will be one column since its or 1 layer.
Now applying the activation function on z gives us
$a^{[1]} = \text{ReLU}(z^{[1]})$
that is $a^{[1]} = \text{ReLU}(W^{[1]}x + b^{[1]})$
This a is the input to the next layer: $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
Since this is the output layer, hence the result from this layer are the prediction by neural network¿
$a^{[3]} = \text{Softmax}(z^{[3]})$
We used Relu in the hidden layers and the softmax in the output layer. Here are the formulas for both

ReLU (Rectified Linear Unit) Activation: f(x)=max(0,x)
Softmax activation function : $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
Output And since we used softmax function at the output layer, the classification output is as follows in the project:
In the project we have 75 unique classes/labels and training dataset size was 192. Therefore the output layer gives 75 probabilities, and we select the one with higest value.

```
    return res

answer = chatbot_response("Computer are good")
```

```
Computer are good
1/1 [==============================] - 0s 54ms/step
[1.99658444e-07 1.40935041e-08 7.44745080e-07 1.52298000e-07
 8.61336602e-08 6.34102544e-05 5.84430927e-06 3.54810777e-06
 1.48899812e-06 4.79055977e-08 4.82589542e-08 1.52826551e-09
 4.52182320e-07 8.15075722e-08 2.95042014e-07 5.01998784e-07
 6.94385704e-08 1.47682954e-07 1.11136148e-06 8.63832295e-01
 9.18136444e-04 1.58126022e-06 2.28370407e-08 5.51832782e-05
 2.66446878e-05 8.27486257e-09 8.00157380e-08 2.04498690e-07
 3.71696984e-08 4.11743576e-06 3.38733486e-07 1.33340448e-01
 3.33691403e-08 3.77157812e-06 2.51370409e-08 2.89254967e-06
 4.83195399e-05 1.68941583e-07 9.22480031e-05 3.12205906e-09
 4.66096526e-06 4.77021104e-07 2.54715218e-07 2.69764826e-07
 1.18130820e-05 1.41783152e-03 1.95830057e-06 5.33229468e-07
 3.88314838e-08 1.68860723e-07 3.10830579e-08 7.08683814e-08
 7.29854861e-08 5.09531917e-10 5.25291775e-08 1.05481968e-05
 2.30053047e-05 6.85530210e-09 2.69362135e-06 1.95282496e-06
 6.92773057e-08 1.17982530e-07 3.26669110e-08 1.58689511e-06
 2.07064232e-07 3.59750203e-08 1.11293764e-06 3.91614492e-07
 2.51230722e-06 1.01805861e-04 1.82969040e-06 8.10923120e-06
 2.10154369e-07 6.07959805e-07 1.03753756e-07]
[{'intent': 'comp', 'probability': '0.8638323'}]
```

Figure 6: Prediction of Classes

1.8 Final Chatbot

Now that our model is trained, we're approaching the final phase: creating the chatbot itself. But, it's not as straightforward as it seems. While the model is ready to classify queries, it operates on numeric data, which means that before seeking predictions, we must preprocess any new query through the same steps we took during training. This includes tokenization, lemmatization, and converting into numeric data, steps applied to the training data. Only then can we feed the processed query into the model for classification. So, the creation of the chatbot involves not just implementing the model, but also integrating the necessary preprocessing steps seamlessly into the chatbot's workflow to ensure accurate predictions.
After processing the query and feeding it into the model, we receive a set of

probabilities, typically representing different classes or labels. To determine the predicted class, we select the one with the highest probability score. Once we've identified the class, we retrieve the corresponding response from our dataset to display as the chatbot's reply. This process ensures that the chatbot provides relevant and accurate responses based on the input query. Below is how the chatbot works:

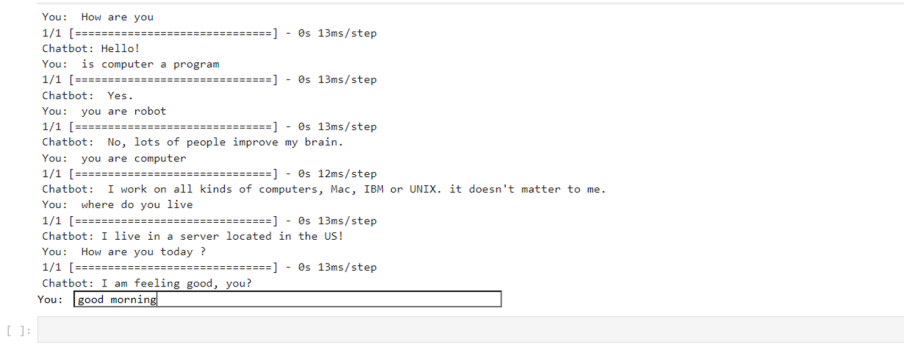

Figure 7: Chatbot

references: Main reference: https://medium.com/@ramakrushnamohapatra/create-an-ai-chatbot-from-scratch-738ea385d108 extra refrences https://handsonai.medium.com/developing-a-simple-chatbot-with-python-and-tensorflow-a-step-by-step-tutorial-0d35767e113b https://towardsdatascience.com/how-to-build-a-basic-chatbot-from-scratch-f63a2ccf5262

1.9 Code

```
import random
from tensorflow.keras.optimizers import SGD
from keras.layers import Dense, Dropout
from keras.models import load_model
from keras.models import Sequential
from sklearn.metrics import confusion_matrix
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
import numpy as np
import pickle
import json
import nltk
from nltk.stem import WordNetLemmatizer
import re

lemmatizer = WordNetLemmatizer()
nltk.download('omw-1.4')
nltk.download("punkt")
nltk.download("wordnet")

# initialize the files
words = [] #saving words after tokenizing them from pattern
tags = []  # tags
TagsAndWords = [] # tags + tokenized words
ignore_words = ["?", "!", ".", "'s", "'", ",","", " "]
data_file = open("/home/d81v711/intents.json").read()
intents = json.loads(data_file)

for intent in intents["intents"]:
    for pattern in intent["patterns"]:

        # take each word and tokenize it
        w = nltk.word_tokenize(pattern)
        words.extend(w)

        # adding tokenized words along with tags
        TagsAndWords.append((w, intent["tag"]))
```

```
        # adding all tags to our tags list
        if intent["tag"] not in tags:
            tags.append(intent["tag"])

#removing single alphabets and digits
print(len(words), "tokenized words", words)
words_new = []

#pattern = pattern = r'\b\w\b'
#words = [re.sub(pattern, '', w) for w in words]

#now that we have tokenized words, we will now lemmatize it
#converting to single form like running to run,
#working to work etc
words = [lemmatizer.lemmatize(w.lower())

for w in words if w not in ignore_words]
# this is our final words list
words = sorted(list(set(words)))
print(len(words), "lemmatized words", words)

tags = sorted(list(set(tags)))

print(len(TagsAndWords), "Tags and words")

print(len(tags), "classes", tags)

# this pickle object idea I took from
pickle.dump(words, open("words.pkl", "wb"))
pickle.dump(tags, open("classes.pkl", "wb"))

#dataset
dataset = {}
#Here we are creating the dataset with unique tags / classes
#dataset will have 2 things 1 words list, second is the tag

for tw in TagsAndWords:

    tag_value = tw[1]
   # print("tag:", tag_value)
    if tag_value in dataset:
        wordsList = dataset[tag_value]
    else:
        wordsList = []
```

```
    # tags and words list has tokenized words from the
    pattern_words = tw[0]

    # Since we lemmatized the words in the words list.
    # Also removed the digits and single alphabets,
    # so will do the same here as well
    pattern_words = [lemmatizer.lemmatize(word.lower())
    for word in pattern_words]
    #pattern = pattern = r'\b\w\b'
    #pattern_words = [re.sub(pattern, '', word)
    for word in pattern_words]

    #But we also removed the words which are in ignore word list.
    for w in words:
        if w in pattern_words:
          #  print("word:" ,w)
            wordsList.append(w)

    dataset[tag_value] = wordsList



for key, value in dataset.items():
    print(key, "->", value)

#Since we have dictionary with different
#length of words, so lets make it numerical and also
#make it of same length
training = []
y_row = [0] * len(tags)
x_row = [0] * len(words)

for key, value in dataset.items():
    output_row = list(y_row)
    output_row[tags.index(key)] = 1

    data_row = list(x_row)
    for w in value:
        data_row[words.index(w)] = 1

    training.append([data_row, output_row])

print("Training dataset size: ", len(training))
```

```
for i in range(2):
    print("Row", i+1)
    print("All unique Words:", training[i][0])
    print("Tag for these words :", training[i][1])
    print()

# shuffle our features and turn into np.array
random.shuffle(training)

# Separate bag-of-words representations and output labels
train_x = [item[0] for item in training]
train_y = [item[1] for item in training]

#print(len(train_y))
# Convert to NumPy arrays
train_x = np.array(train_x)
train_y = np.array(train_y)
print("Training data created")

from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test =
train_test_split(train_x, train_y, test_size=0.2, random_state=42)

model = Sequential()
model.add(Dense(128, input_shape=(len(train_x[0]),), activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(64, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(len(train_y[0]), activation="softmax"))
model.summary()

# Compile the model
sgd = SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
model.compile(loss="categorical_crossentropy",
optimizer=sgd, metrics=["accuracy"])

# fitting and saving the model
hist = model.fit(np.array(train_x), np.array(train_y),
epochs=200, batch_size=5, verbose=1)

# Evaluate the model on the training data
train_loss, train_accuracy = model.evaluate(x_train, y_train)
print("Training Accuracy:", train_accuracy)

# Evaluate the model on the testing data
```

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Testing Accuracy:", test_accuracy)

model.save("chatbot_model.h5", hist)
print("model created")

#load test file
wordsT= [] #words have tpkenized words from pattern
classesT= []  # tags
documentsT= [] # tags + tokenized words
ignore_words = ["?", "!"]
data_fileTest = open("/home/d81v711/intents_test.json").read()
intents_test = json.loads(data_fileTest)
testing = []

#make bag of words for the patterns in the test file.
for inte in intents_test["intents"]:
    for patt in inte["patterns"]:

        # take each word and tokenize it
        w = nltk.word_tokenize(patt)
        wordsT.extend(w)
        # adding documents
        documentsT.append((w, inte["tag"]))

        # adding classes to our class list
        if inte["tag"] not in classesT:
            classesT.append(inte["tag"])

output_empty = [0] * len(classes)
for doc in documentsT:
    # initializing bag of words
    bag = []
    # list of tokenized words for the pattern
    pattern_wordsT = doc[0]
    # lemmatize each word - create base word, in
    # attempt to represent related words
    pattern_wordsT = [lemmatizer.lemmatize(word.lower())
    for word in pattern_wordsT]
    # create our bag of words array with 1,
    # if word match found in current pattern
    for w in words:
        if w in pattern_wordsT:
            #print("word:" ,w)
            bag.append(1)
        else:
```

```
            bag.append(0)

    # output is a '0' for each tag and '1'
    # for current tag (for each pattern)
    output_row = list(output_empty)
    output_row[classes.index(doc[1])] = 1


    #print("Tag for this word:",tag_value)
    testing.append([bag, output_row])


print("Testing dataset size: ", len(testing))

for i in range(2):
    print("Row", i+1)
    print("Bag of Words:", testing[i][0])
    print("Output Row:", testing[i][1])
    print()


# Separate bag-of-words representations and output labels
test_x = [item[0] for item in testing]
test_y = [item[1] for item in testing]

#print(len(train_y))
# Convert to NumPy arrays
test_x = np.array(test_x)
test_y = np.array(test_y)

print("Testing data created")

# chat functionalities
def tokenize_sentence(sentence):
    sentence_words = nltk.word_tokenize(sentence)
    sentence_words = [lemmatizer.lemmatize(word.lower())
    for word in sentence_words]
    return sentence_words


# Fromt the question, we need to find
# words which match with our unique words (dictionary /list).
# We are converting the setence words
# into the same form which our model recognize / is trained on

def compatible_words(sentence, words):
```

```
    # tokenize and lemmatize the pattern
    sentence_words = tokenize_sentence(sentence)
    # bag of words - matrix of N words, vocabulary matrix
    bag = [0] * len(words)
    for s in sentence_words:
        for i, w in enumerate(words):
            if w == s:
                # assign 1 if current word
                # is in the vocabulary position
                bag[i] = 1
    return np.array(bag)

def predict_class(sentence, model):
    # filter out predictions below a threshold
    cwords = compatible_words(sentence, words)

    input_features = np.array([cwords])
    print("Input sentence:", sentence)
    response = model.predict(input_features)[0]
    print("Prediction probabilities:", response)
    ERROR_THRESHOLD = 0.25

    results = [[i, r] for i, r in enumerate(response)
    if r > ERROR_THRESHOLD]

    # sort by strength of probability
    results.sort(key=lambda x: x[1], reverse=True)

    return_list = []
    for r in results:
        return_list.append({"intent": classes[r[0]],
        "probability": str(r[1])})

    print("Predicted intents:", return_list)
    return return_list


def getResponse(ints, intents_json):
    if not ints:
        return "I am sorry, I do not have answer to this question"
    tag = ints[0]["intent"]
    list_of_intents = intents_json["intents"]
    for i in list_of_intents:
        if i["tag"] == tag:
            result = random.choice(i["responses"])
```

```
            break
    return result

def chatbot_response(question):
    ints = predict_class(question, model)
    res = getResponse(ints, intents)
    return res

# Start chatbot
while True:
    question = input('You: ')
    answer = chatbot_response(question)
    print('Chatbot:', answer)
```