Chatbot in Python and Flask

#python#machinelearning#ai

Perhaps you have heard this term and wondered: what is this chatbot, what is it used for, do I really need one, how can I create one? If you just want to build your own simple chatbot, this article will take you through all the steps in creating one for yourself. Let's dive right in.

Preliminaries:

- Python programming knowledge
- Flask framework knowledge
- Machine learning and algorithm knowledge

Table of Contents

What is a Chatbot?

A Chatbot, also called an Artificial chat agent, is a software program driven by machine learning algorithms that aim at simulating a human-human like conversation with a user by either taking input as text or speech from the user.

Where is it used?

Chatbots have extensive usage, and we can not expound on all the possibilities where it can be of use. But basically, you'll find them in: Help desks, transaction processing, customer support, booking services, and providing 24-7 real-time chat with clients.

Do I need one?

Well, this is a personalized opinion where one has to do a cost-benefit analysis and decide whether it is a worthwhile project.

At the current technology stand, most companies are slowly transitioning to use chatbots for their in-demand day-day services. A good example that everybody uses is the Google Assistant, Apple Siri, Samsung Bixby, and Amazon Alexa.
In this article, we will learn how to create one in Python using TensorFlow to train the model and Natural Language Processing(nltk) to help the machine understand user queries.
There are two broad categories of chatbots:

1. **Rule-Based approach** - Here the bot is trained based on some set rules. It is from these rules that the bot can process simple queries but can fail to process complex ones.
2. **Self-Learning approach** - Here the bot uses some machine learning algorithms and techniques to chat. It is further subcategorized into two:
    - **Retrieval-Based models** - In this model, the bot retrieves the best response from a list depending on the user input.
    - **Generative models** - This model comes up with an answer rather than searching from a given list. These are the Intelligent Bots.

In this chatbot, we will use the rule-based approach.

Terms to encounter

Natural Language Processing(nltk) - This is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages.
Lemmatization - This is the process of grouping together the different inflected forms of a word so they can be analyzed as a single item and is a variation of stemming. For example “feet” and “foot” are both recognized as “foot”.
Stemming - This is the process of reducing inflected words to their word stem, base, or root form. For example, if we were to stem the word “eat”, “eating”, “eats”, the result would be the single word “eat”.
Tokenization - *Tokens* are individual words and *“tokenization”* is taking a text or set of text and breaking it up into its individual words or sentences.
Bag of Words - This is an NLP technique of text modeling for representing text data for machine learning algorithms. It is a way of extracting features from the text for use in machine learning algorithms.

Let's start programming now

Directory structure

```
|--  Static
|    |--  style.css
|--  Templates
|    |--  index.html
|--  app.py
|--  train.py
|--  intents.json
|--  words.pkl
|--  classes.pkl
|--  chatbot_model.h5
```

First, we need to make sure that we have all the required libraries and modules. Install tensorflow, nltk, and flask using the following command.

```
pip install tensorflow
pip install tensorflow-gpu
pip install nltk
pip install Flask
```

Our intents.json file should look something like this: Here I'm sorry it's quite a long file I can't embed it here.
Now that we have everything ready, Let's start by creating the train.py file. We need to import all the packages and libraries that we're going to be using.

```
# libraries
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
import numpy as np
import pickle
import json
import nltk
from nltk.stem import WordNetLemmatizer
```

Download **punkt**, **omw-1.4**, and **wordnet** package. Punkt is a pre-trained tokenizer model for the English language that divides the text into a list of sentences.

```
nltk.download('omw-1.4')
nltk.download("punkt")
nltk.download("wordnet")
```

We now need to initialize some files and load our training data. Note that we are going to be ignoring "?" and "!". If you have some other symbols or letters that you want the model to ignore you can add them at the ignore_words array.

```
# init file
words = []
classes = []
documents = []
ignore_words = ["?", "!"]
data_file = open("intents.json").read()
intents = json.loads(data_file)
```

Then we tokenize our words

```
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 documents
        documents.append((w, intent["tag"]))

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

Next, we shall lemmatize the words and dump them in a pickle file

```
words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in ignore_words]
words = sorted(list(set(words)))

classes = sorted(list(set(classes)))

print(len(documents), "documents")

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

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

```
pickle.dump(words, open("words.pkl", "wb"))
pickle.dump(classes, open("classes.pkl", "wb"))
```

Now that we have our training data in place, we initialize the model training

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

    training.append([bag, output_row])
# shuffle our features and turn into np.array
random.shuffle(training)
training = np.array(training)
# create train and test lists. X - patterns, Y - intents
train_x = list(training[:, 0])
train_y = list(training[:, 1])
print("Training data created")
```

We shall create a 3-layer output model. The first layer having 128 neurons, the second layer having 64 neurons, and the third layer contains the number of neurons equal to the number of intents to predict output intent with softmax. We shall be using ReLu activation function as it's easier to train and achieves good perfomance.

```
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()
```

Then, we compile the model. Stochastic gradient descent with Nesterov accelerated gradient gives good results for this model. I won't go into details about Stochastic gradient descent as this is a vastly complicated topic on its own.

```
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss="categorical_crossentropy", optimizer=sgd, metrics=["accuracy"])
```

OPTIONAL: For choosing an optimal number of training epochs to avoid underfitting or overfitting use an early stopping callback to Keras based on either accuracy or loss monitoring. If the loss is being monitored, training comes to halt when there is an increment observed in loss values. Or, If accuracy is being monitored, training comes to halt when there is a decrement observed in accuracy values.

```
from Keras import callbacks
 earlystopping = callbacks.EarlyStopping(monitor ="loss", mode ="min", patience = 5,
restore_best_weights = True)
callbacks =[earlystopping]
```

Now we can train our model and save it for fast access from the Flask REST API without the need of retraining.

```
hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1)
model.save("chatbot_model.h5", hist)
print("model created")
```

Your train.py file now should look like this:

```
# libraries

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
```

```
import numpy as np
import pickle
import json
import nltk
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
nltk.download('omw-1.4')
nltk.download("punkt")
nltk.download("wordnet")
# init file
words = []
classes = []
documents = []
ignore_words = ["?", "!"]
data_file = open("intents.json").read()
intents = json.loads(data_file)
```

```
# words

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 documents

        documents.append((w,
intent["tag"]))




        # adding classes to our class
list

        if intent["tag"] not in classes:

            classes.append(intent["tag"])




# lemmatizer

words = [lemmatizer.lemmatize(w.lower())
for w in words if w not in ignore_words]

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

```
classes = sorted(list(set(classes)))
print(len(documents), "documents")
print(len(classes), "classes", classes)
print(len(words), "unique lemmatized words", words)
pickle.dump(words, open("words.pkl", "wb"))
pickle.dump(classes, open("classes.pkl", "wb"))
# initializing training data
training = []
output_empty = [0] * len(classes)
for doc in documents:
```

```
    # initializing bag of words

    bag = []

    # list of tokenized words for the
pattern

    pattern_words = doc[0]

    # lemmatize each word - create base
word, in attempt to represent related
words

    pattern_words =
[lemmatizer.lemmatize(word.lower()) for
word in pattern_words]

    # create our bag of words array with
1, if word match found in current
pattern

    for w in words:

        bag.append(1) if w in
pattern_words 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




    training.append([bag, output_row])
```

```
# shuffle our features and turn into np.array

random.shuffle(training)

training = np.array(training)

# create train and test lists. X - patterns, Y - intents

train_x = list(training[:, 0])

train_y = list(training[:, 1])

print("Training data created")




# actual training

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 model. Stochastic gradient descent with Nesterov accelerated gradient gives good results for this model

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

model.compile(loss="categorical_crossentropy", optimizer=sgd, metrics=["accuracy"])



#Optional snippet - I have commented it as I did not use it.



# from keras import callbacks

# earlystopping = callbacks.EarlyStopping(monitor ="loss", mode ="min", patience = 5, restore_best_weights = True)

# callbacks =[earlystopping]



# fitting and saving the model

hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1)

model.save("chatbot_model.h5", hist)
```

```
print("model created")
```

view raw

train.py hosted with ❤ by GitHub

Run the train.py to create the model.

Now that we are done with training let's create the Flask interface to initialize the chat functionalities.

We load the required libraries and initialize the Flask app

```
# libraries
import random
import numpy as np
import pickle
import json
from flask import Flask, render_template, request
from flask_ngrok import run_with_ngrok
import nltk
from keras.models import load_model
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()


# chat initialization
model = load_model("chatbot_model.h5")
intents = json.loads(open("intents.json").read())
words = pickle.load(open("words.pkl", "rb"))
classes = pickle.load(open("classes.pkl", "rb"))

app = Flask(__name__)
#run_with_ngrok(app) -Use this option if you have ngrok and you want to expose your chatbot to the real world

@app.route("/")
def home():
    return render_template("index.html")
```

This function will be called every time a user sends a message to the chatbot and returns a corresponding response based on the user query.

```
@app.route("/get", methods=["POST"])
def chatbot_response():
    msg = request.form["msg"]
    if msg.startswith('my name is'):
```

```
        name = msg[11:]
        ints = predict_class(msg, model)
        res1 = getResponse(ints, intents)
        res =res1.replace("{n}",name)
    elif msg.startswith('hi my name is'):
        name = msg[14:]
        ints = predict_class(msg, model)
        res1 = getResponse(ints, intents)
        res =res1.replace("{n}",name)
    else:
        ints = predict_class(msg, model)
        res = getResponse(ints, intents)
    return res
```

The above function will call the following functions which clean up sentences and return a bag of words based on the user input.

```
def clean_up_sentence(sentence):
    sentence_words = nltk.word_tokenize(sentence)
    sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words]
    return sentence_words


# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence
def bow(sentence, words, show_details=True):
    # tokenize the pattern
    sentence_words = clean_up_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
                if show_details:
                    print("found in bag: %s" % w)
    return np.array(bag)
```

The next functions are for predicting the response to give to the user where they fetch that response from the chatbot_model.h5 file generated after the training.

```
def predict_class(sentence, model):
    # filter out predictions below a threshold
    p = bow(sentence, words, show_details=False)
```

```
    res = model.predict(np.array([p]))[0]
    ERROR_THRESHOLD = 0.25
    results = [[i, r] for i, r in enumerate(res) 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])})
    return return_list


def getResponse(ints, intents_json):
    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
```

Generally, our app.py should look like this:

```
# libraries

import random

import numpy as np

import pickle

import json

from flask import Flask,
render_template, request

from flask_ngrok import run_with_ngrok

import nltk

from keras.models import load_model

from nltk.stem import WordNetLemmatizer
```

```
lemmatizer = WordNetLemmatizer()
```

```
# chat initialization
```

```
model = load_model("chatbot_model.h5")
```

```
intents =
json.loads(open("intents.json").read())
```

```
words = pickle.load(open("words.pkl",
"rb"))
```

```
classes =
pickle.load(open("classes.pkl", "rb"))
```

```
app = Flask(__name__)
```

```
@app.route("/")
```

```
def home():
```

```
   return render_template("index.html")
```

```
@app.route("/get", methods=["POST"])
```

```
def chatbot_response():
    msg = request.form["msg"]
    #checks is a user has given a name,
in order to give a personalized feedback
    if msg.startswith('my name is'):
        name = msg[11:]
        ints = predict_class(msg, model)
        res1 = getResponse(ints, intents)
        res =res1.replace("{n}",name)
    elif msg.startswith('hi my name is'):
        name = msg[14:]
        ints = predict_class(msg, model)
        res1 = getResponse(ints, intents)
        res =res1.replace("{n}",name)
    #if no name is passed execute
normally
    else:
        ints = predict_class(msg, model)
        res = getResponse(ints, intents)
    return res
```

```
# chat functionalities

def clean_up_sentence(sentence):
    sentence_words = nltk.word_tokenize(sentence)
    sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words]
    return sentence_words

# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence

def bow(sentence, words, show_details=True):
    # tokenize the pattern
    sentence_words = clean_up_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
                if show_details:
                    print("found in bag: %s" % w)
    return np.array(bag)
```

```
def predict_class(sentence, model):
    # filter out predictions below a threshold
    p = bow(sentence, words, show_details=False)
    res = model.predict(np.array([p]))[0]
    ERROR_THRESHOLD = 0.25
    results = [[i, r] for i, r in enumerate(res) 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])})

    return return_list




def getResponse(ints, intents_json):

    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
```

```
if __name__ == "__main__":

    app.run()
```

view raw

app.py hosted with ❤ by GitHub

Lastly, we have the index.html file which I won't go into detail about as it's basic HTML and CSS.

```
<!DOCTYPE html>

<html>

<head>

    <link rel="stylesheet"
type="text/css" href="{{
url_for('static',
filename='style.css')}}" />

    <!-- <link rel="stylesheet"
type="text/css" href="{{
url_for('static', filename='css.css')}}"
/> -->

    <script
src="https://ajax.googleapis.com/ajax/li
bs/jquery/3.2.1/jquery.min.js"></script>

</head>
```

```
<body>

   <div class="row">

       <div class="col-md-10 mr-auto
ml-auto">

   <h1>Dentrice AI ChatBot</h1>

   <form>

       <div id="chatbox">

           <div class="col-md-8 ml-auto
mr-auto">

               <p
class="botText"><span>Hi! I'm Your
bot.</span></p>

           </div>

       </div>

       <div id="userInput" class="row">

           <div class="col-md-10">

               <input id="text"
type="text" name="msg"
placeholder="Message"
class="form-control">

               <button type="submit"
id="send" class="btn
btn-warning">Send</button>

           </div>
```

```
        </div>
    </form>
</div>
</div>



<script>
    $(document).ready(function() {
        $("form").on("submit",
function(event) {
            var rawText =
$("#text").val();
            var userHtml = '<p
class="userText"><span>' + rawText +
"</span></p>";
            $("#text").val("");

$("#chatbox").append(userHtml);

document.getElementById("userInput").scr
ollIntoView({
                block: "start",
                behavior: "smooth",
            });
```

```
            $.ajax({
                data: {
                    msg: rawText,
                },
                type: "POST",
                url: "/get",
            }).done(function(data) {
                var botHtml = '<p class="botText"><span>' + data + "</span></p>";

$("#chatbox").append($.parseHTML(botHtml));

document.getElementById("userInput").scrollIntoView({
                    block: "start",
                    behavior: "smooth",
                });
            });
            event.preventDefault();
        });
    });
```

	</script>
	</body>
	</html>