APPENDIX

```
import nltk
from nltk.stem import WordNetLemmatizer
# from nltk.corpus import stopwords
from numpy.lib.twodim_base import triu_indices_from
lemmatizer = WordNetLemmatizer()
import json
import pickle

import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, LSTM
from keras.optimizers import SGD
import random

nltk.download('punkt')
nltk.download('wordnet')
# nltk.download('stopwords')

words=[]
classes = []
documents = []
ignore_words = ['?', '!']
data_file = open('futbot.json', encoding='utf-8').read()

intents = json.loads(data_file)
print('Intent file read successfully')

# setting stop words
# stop_words = set(stopwords.words('english'))

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

        #tokenize each word
        w = nltk.word_tokenize(pattern)
        words.extend(w)
        #add documents in the corpus
        documents.append((w, intent['tag']))

        # add to our classes list
        if intent['tag'] not in classes:
            classes.append(intent['tag'])


# # lemmaztize and lower each word and remove duplicates
words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in
ignore_words]
```

```
words = sorted(list(set(words)))
print('word sorted...')
print('classes are: ', classes)



# sort classes
print('classes length: ', len(classes))

classes = sorted(set(classes))
print('class sorted')
# documents = combination between patterns and intents
print (len(documents), "documents")
# classes = intents
print (len(classes), "classes", classes)
# words = all words, vocabulary
print (len(words), "unique lemmatized words", words)


pickle.dump(words,open('texts.pkl','wb'))
print('texts.pkl dumped successfully')
pickle.dump(classes,open('labels.pkl','wb'))
print('lebel.pkl dumped successfully')

# create our training data
training = []
# create an empty array for our output
output_empty = [0] * len(classes)
# training set, bag of words for each sentence
for doc in documents:
    # initialize our 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")


# Create model - 3 layers. First layer 128 neurons, second layer 64 neurons
and 3rd output layer contains number of neurons
# equal to number of intents to predict output intent with softmax
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(Dense(64, activation='relu'))

model.add(Dense(len(train_y[0]), activation='softmax'))

# 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='adam',
metrics=['accuracy'])
# sgd
#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('model.h5', hist)

# # Define the LSTM model architecture
# seq_length = len(train_x[0])
# model = Sequential()
# model.add(LSTM(units=64, input_shape=(seq_length, 1),
return_sequences=True))
# model.add(Dropout(0.2))
# model.add(LSTM(units=64))
# model.add(Dropout(0.2))
# model.add(Dense(units=seq_length, activation='softmax'))

# model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# # Train the model
# hist = model.fit(np.array(train_x), np.array(train_y), epochs=200,
batch_size=5, verbose=1)
# model.save('model.h5', hist)
```

```
print("model created")



from flask import Flask, render_template, request, Response
import random
import json
from keras.models import load_model
import numpy as np
import pickle
from nltk.stem import WordNetLemmatizer
import nltk
from nltk.corpus import stopwords

import speech_recognition as sr
import pyttsx3
from txt2speech import speak_text, mic

nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')

nltk.download('popular')
lemmatizer = WordNetLemmatizer()



model = load_model('model.h5')
intents = json.loads(open('futbot.json', encoding='utf-8').read())
words = pickle.load(open('texts.pkl', 'rb'))
classes = pickle.load(open('labels.pkl', 'rb'))


# setting stop words
stop_words = set(stopwords.words('english'))

def clean_up_sentence(sentence):
    # tokenize the pattern - split words into array
    sentence_words = nltk.word_tokenize(sentence)
    # stem each word - create short form for word
    sentence_words = [lemmatizer.lemmatize(word.lower()) for word in
sentence_words]
    # words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in
stop_words]if word not in stop_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):
    if (len(ints) < 1):
        return "Can you please rephrase your question?, I'm having trouble
understanding"
    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(msg):
    ints = predict_class(msg, model)
    res = getResponse(ints, intents)
    print(res)
    return res
```

```
app = Flask(__name__)
app.static_folder = 'static'


@app.route("/")
def home():

    return render_template("index.html")

wlcm_txt ="Hey dear welcome, I am your virtual assistant what can i do for
you? Note,You can either speek or write your querry! and also, You must
confirm what ever solution is given to you as i am only a bot."
speak_text(wlcm_txt)

@app.route("/get")
def get_bot_response():
    userText = request.args.get('msg')
    response = chatbot_response(userText)
    speak_text(response)
    # return Response(json.dumps({'res' : response}),
mimetype='application/json');
    return response


@app.route("/bot/api/v1/prompt", methods=['POST'])
def get_api_response():
    userText = request.args.get('msg')
    response = chatbot_response(userText);
    return Response(json.dumps({'res' : response}),
mimetype='application/json');

if __name__ == "__main__":
    app.run(debug=True)



import speech_recognition as sr
import pyttsx3

# Initialize the recognizer
r = sr.Recognizer()

# Function to convert text to speech
def speak_text(command):
    # Initialize the engine
    engine = pyttsx3.init()

    """VOICE"""
    voices = engine.getProperty('voices')       #getting details of current
voice
    #engine.setProperty('voice', voices[0].id)  #changing index, changes
voices. o for male
    engine.setProperty('voice', voices[1].id)   #changing index, changes
voices. 1 for female
    engine.say(command)
    engine.runAndWait()

# Loop infinitely for user to speak
def mic(duration=0.2):
    while True:
        try:
            # use the microphone as source for input.
            with sr.Microphone() as source:
                # wait for a second to let the recognizer
                # adjust the energy threshold based on
                # the surrounding noise level
                r.adjust_for_ambient_noise(source, duration)

                # listens for the user's input
                audio = r.listen(source)

                # Using google to recognize audio
                MyText = r.recognize_google(audio)
                MyText = MyText.lower()

                print("Did you say: ", MyText)
                speak_text(MyText)

        except sr.RequestError as e:
            print("Could not request results; {0}".format(e))

        except sr.UnknownValueError:
            print("unknown error occurred")


if __name__ == "__main__":
    # Start speaking
    speak_text("Hello! my name is Jenifa... Go ahead and say your querry or
type it. You know what? just do as you wish")

    # Start listening
    # mic()
```