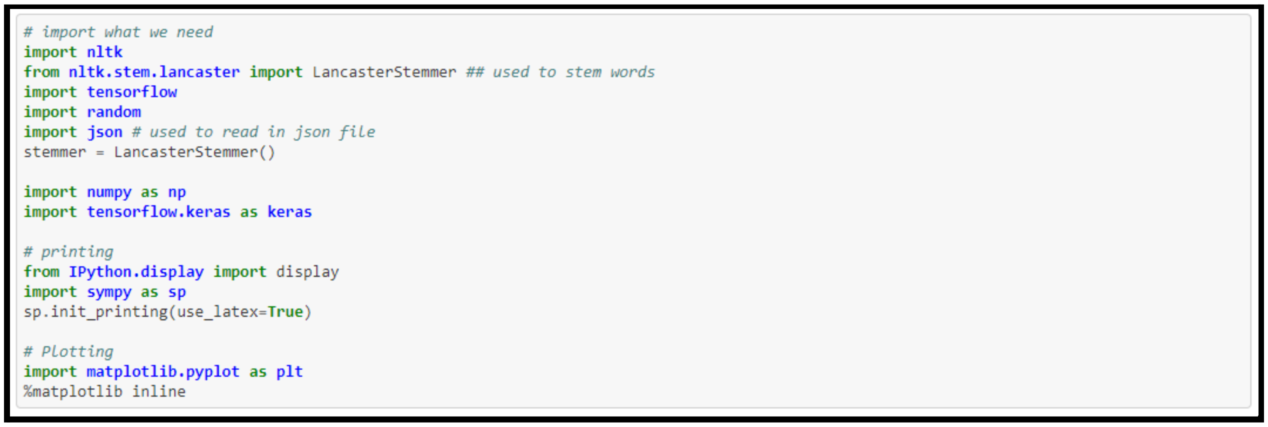
Python Chatbot Demo

CSCI 4850-5850 - Neural Networks

Lorenzo McDaniel, Paula Pamplona, David Brugger, Brooke Bound, Jake Hagenow

Step 1: Install Necessary Packages

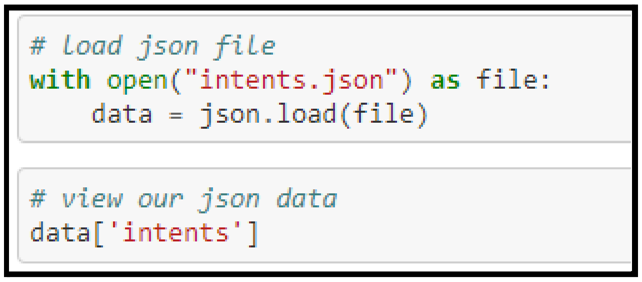
Firstly, we need to install the tools that we used to create this chatbot. We used a number of tools, including nltk, tensorflow, random, json, numpy, keras, sympy, and matplotlib.



```
# import what we need
import nltk
from nltk.stem.lancaster import LancasterStemmer ## used to stem words
import tensorflow
import random
import json # used to read in json file
stemmer = LancasterStemmer()

import numpy as np
import tensorflow.keras as keras

# printing
from IPython.display import display
import sympy as sp
sp.init_printing(use_latex=True)

# Plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Step 2: Import and View Intents (data)

Now we need to pull our data into the chatbot. In order to do this, you will need to have all data saved in a file called “intents.json” in the same directory as your chatbot model. To open and view this data, run the following cells:

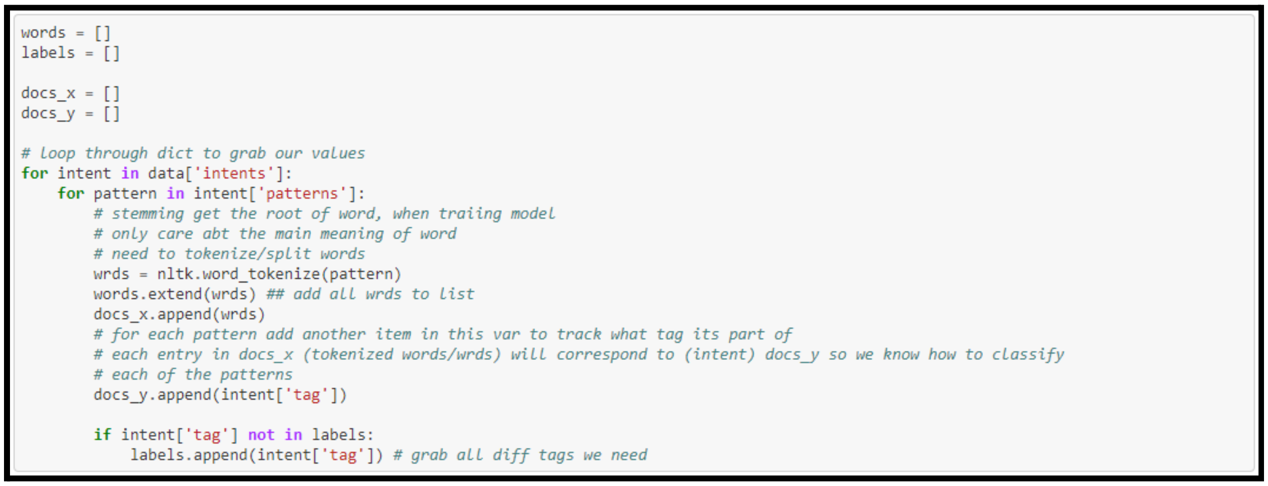


```
# load json file
with open("intents.json") as file:
    data = json.load(file)
```

```
# view our json data
data['intents']
```

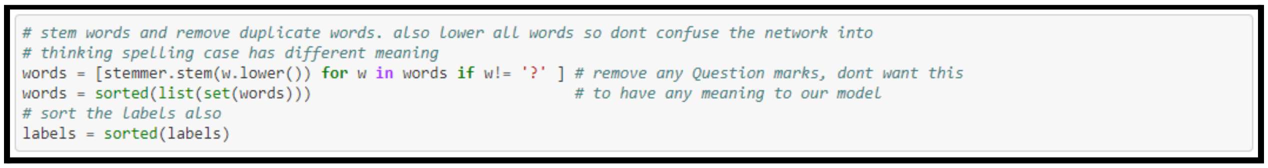
Step 3: Prepare Data for Training

The chatbot is going to be unable to be trained on the raw data alone, so we have to trim the data and modify it a bit before we dive into the training of the model. In order to make the data usable for the bot, we need to stem the data in order for it to be readable. The model only cares about the roots of words, so we need to tokenize/split each word and add them to lists.



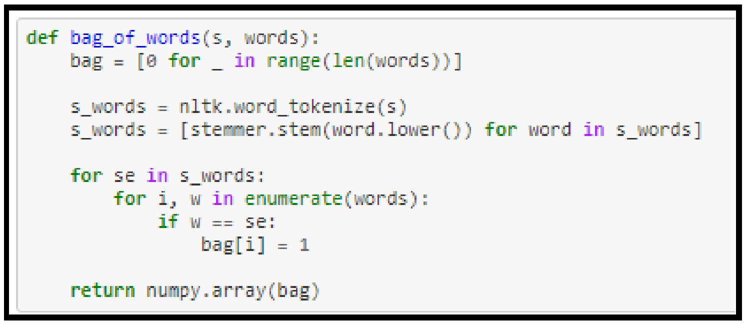
```python
words = []
labels = []

docs_x = []
docs_y = []

# loop through dict to grab our values
for intent in data['intents']:
    for pattern in intent['patterns']:
        # stemming get the root of word, when traiing model
        # only care abt the main meaning of word
        # need to tokenize/split words
        wrds = nltk.word_tokenize(pattern)
        words.extend(wrds) ## add all wrds to list
        docs_x.append(wrds)
        # for each pattern add another item in this var to track what tag its part of
        # each entry in docs_x (tokenized words/wrds) will correspond to (intent) docs_y so we know how to classify
        # each of the patterns
        docs_y.append(intent['tag'])

        if intent['tag'] not in labels:
            labels.append(intent['tag']) # grab all diff tags we need
```

We then need to remove duplicates and change all letters to lowercase to lessen the odds that the bot gets confused. Finally, we need to remove all question marks, because they do not add any meaning to our bot.

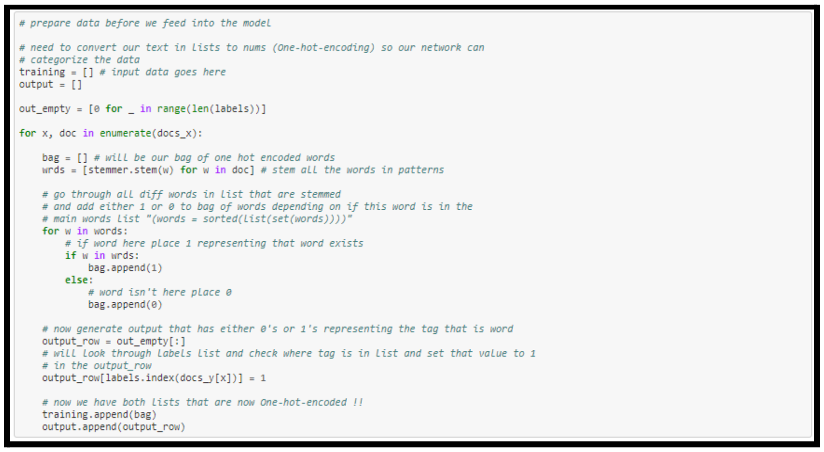


```python
# stem words and remove duplicate words. also lower all words so dont confuse the network into
# thinking spelling case has different meaning
words = [stemmer.stem(w.lower()) for w in words if w!= '?' ] # remove any Question marks, dont want this
words = sorted(list(set(words)))                             # to have any meaning to our model
# sort the labels also
labels = sorted(labels)
```

Lasty, we need to perform one-hot-encoding on the text so that we can categorize data with the bot. We do this by going through the text and tag them with either a 1 or 0 depending on whether or not the word is in our “main” list of words. We then need to go through and fill an output list with 1s and 0s based on our encoded words. Finally, we append our training data and output data with the one-hot-encoded data. In order to use this code, you will need to write an additional function called bag_of_words that works to aid in the tokenization process of one-hot-encoding.

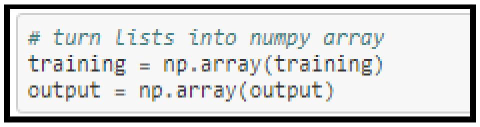


```
def bag_of_words(s, words):
    bag = [0 for _ in range(len(words))]

    s_words = nltk.word_tokenize(s)
    s_words = [stemmer.stem(word.lower()) for word in s_words]

    for se in s_words:
        for i, w in enumerate(words):
            if w == se:
                bag[i] = 1

    return numpy.array(bag)
```



```
# prepare data before we feed into the model

# need to convert our text in lists to nums (One-hot-encoding) so our network can
# categorize the data
training = [] # input data goes here
output = []

out_empty = [0 for _ in range(len(labels))]

for x, doc in enumerate(docs_x):

    bag = [] # will be our bag of one hot encoded words
    wrds = [stemmer.stem(w) for w in doc] # stem all the words in patterns

    # go through all diff words in list that are stemmed
    # and add either 1 or 0 to bag of words depending on if this word is in the
    # main words list "(words = sorted(list(set(words))))"
    for w in words:
        # if word here place 1 representing that word exists
        if w in wrds:
            bag.append(1)
        else:
            # word isn't here place 0
            bag.append(0)

    # now generate output that has either 0's or 1's representing the tag that is word
    output_row = out_empty[:]
    # will look through labels list and check where tag is in list and set that value to 1
    # in the output_row
    output_row[labels.index(docs_y[x])] = 1

    # now we have both lists that are now One-hot-encoded !!
    training.append(bag)
    output.append(output_row)
```

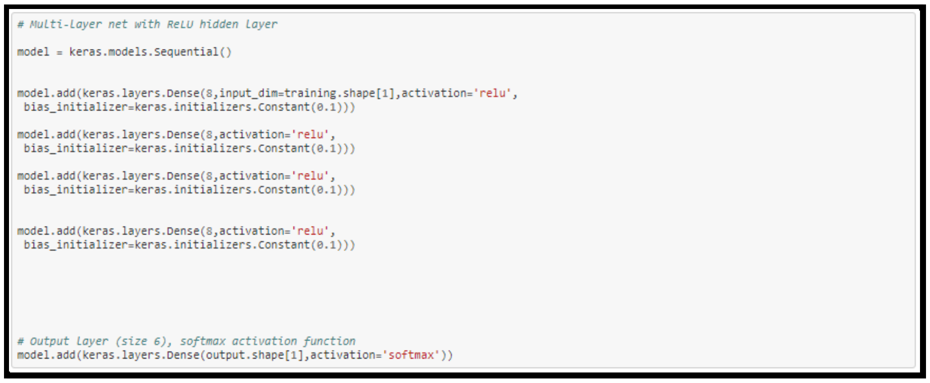
The very last thing we need to do before we turn our focus from our data to our model is to turn the training and output lists into numpy arrays. This is quite simple. Create a new cell, and insert the following code:



```
# turn lists into numpy array
training = np.array(training)
output = np.array(output)
```

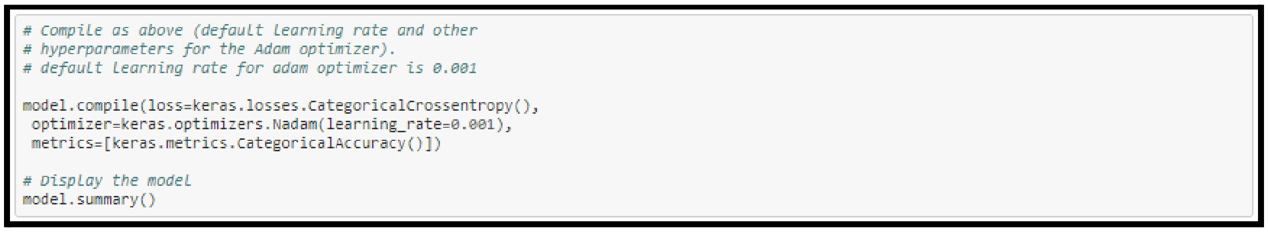
Step 4: Creating and Compiling the Model

Now we create a multi-layer network with a ReLU hidden layer. We are going to make it a Sequential model, with 4 layers, all using ReLU activation functions, and one softmax output layer of size 6.



```
# Multi-Layer net with ReLU hidden layer

model = keras.models.Sequential()


model.add(keras.layers.Dense(8,input_dim=training.shape[1],activation='relu',
 bias_initializer=keras.initializers.Constant(0.1)))

model.add(keras.layers.Dense(8,activation='relu',
 bias_initializer=keras.initializers.Constant(0.1)))

model.add(keras.layers.Dense(8,activation='relu',
 bias_initializer=keras.initializers.Constant(0.1)))


model.add(keras.layers.Dense(8,activation='relu',
 bias_initializer=keras.initializers.Constant(0.1)))





# Output layer (size 6), softmax activation function
model.add(keras.layers.Dense(output.shape[1],activation='softmax'))
```

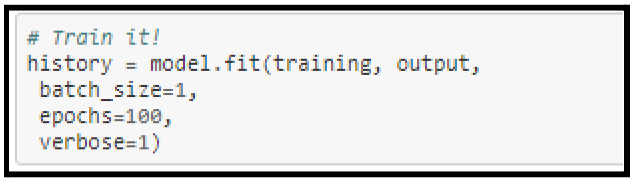
Now that the model has been built, we need to compile it. We are going to use the Adam optimizer, with all hyperparameters default for the Adam optimizer. We also want to be able to visualize the results, so we will be using Categorical Accuracy for the metrics parameter.



```
# Compile as above (default learning rate and other
# hyperparameters for the Adam optimizer).
# default learning rate for adam optimizer is 0.001

model.compile(loss=keras.losses.CategoricalCrossentropy(),
 optimizer=keras.optimizers.Nadam(learning_rate=0.001),
 metrics=[keras.metrics.CategoricalAccuracy()])

# Display the model
model.summary()
```

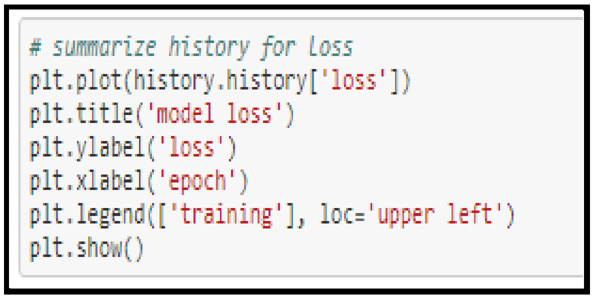
Step 5: Training and Results

Next step is to train the model. We found that a batch size of 1 with 100 epochs was sufficient for the model (note: verbose may be set to 0 to avoid extra content on the screen).

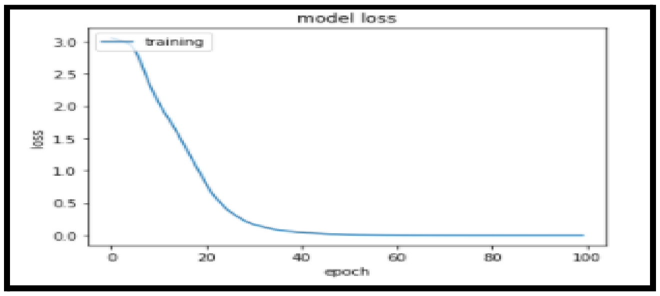


```
# Train it!
history = model.fit(training, output,
 batch_size=1,
 epochs=100,
 verbose=1)
```

After training is complete, it is time to visualize our results. Because we used matplotlib, we can create a graph to show us our model's loss:

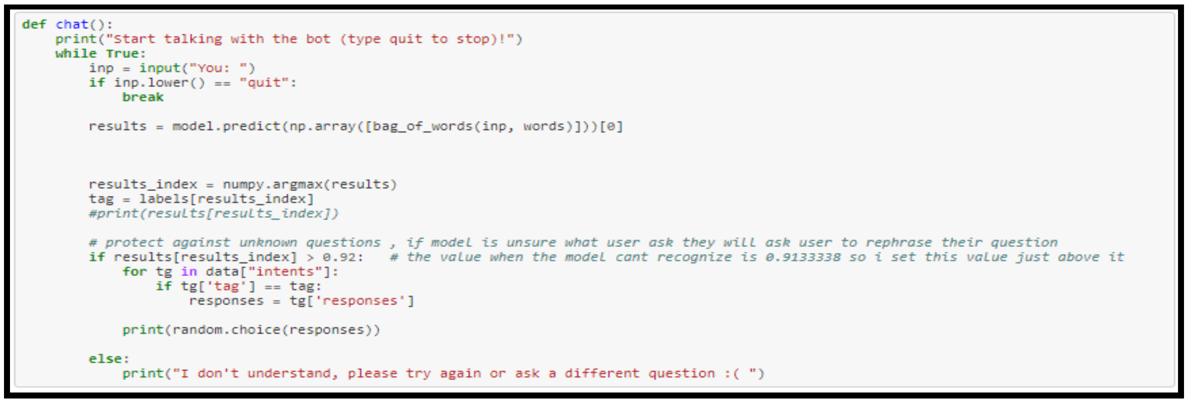


```
# summarize history for loss
plt.plot(history.history['loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['training'], loc='upper left')
plt.show()
```



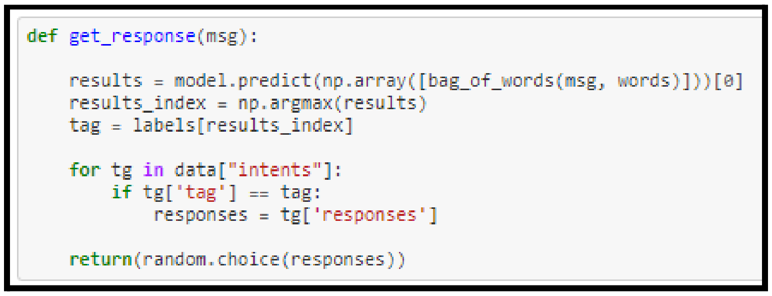
Step 6: Let’s Chat

Finally, let’s create our chat function, which will allow us to actually have a conversation with our bot. We are going to be creating a while loop that will allow for conversation to continue until the user types “quit.” As an added measure of protection from replies that do not make sense, if the bot has low confidence in any input from the user, it will simply ask the user to rephrase the question or ask a different one.

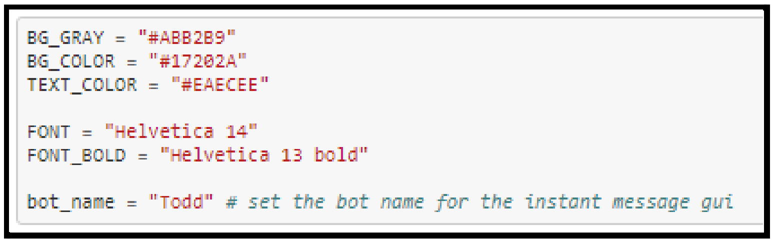


```python
def chat():
    print("Start talking with the bot (type quit to stop)!")
    while True:
        inp = input("You: ")
        if inp.lower() == "quit":
            break

        results = model.predict(np.array([bag_of_words(inp, words)]))[0]



        results_index = numpy.argmax(results)
        tag = labels[results_index]
        #print(results[results_index])

        # protect against unknown questions , if model is unsure what user ask they will ask user to rephrase their question
        if results[results_index] > 0.92:   # the value when the model cant recognize is 0.9133338 so i set this value just above it
            for tg in data["intents"]:
                if tg['tag'] == tag:
                    responses = tg['responses']

            print(random.choice(responses))

        else:
            print("I don't understand, please try again or ask a different question :( ")
```

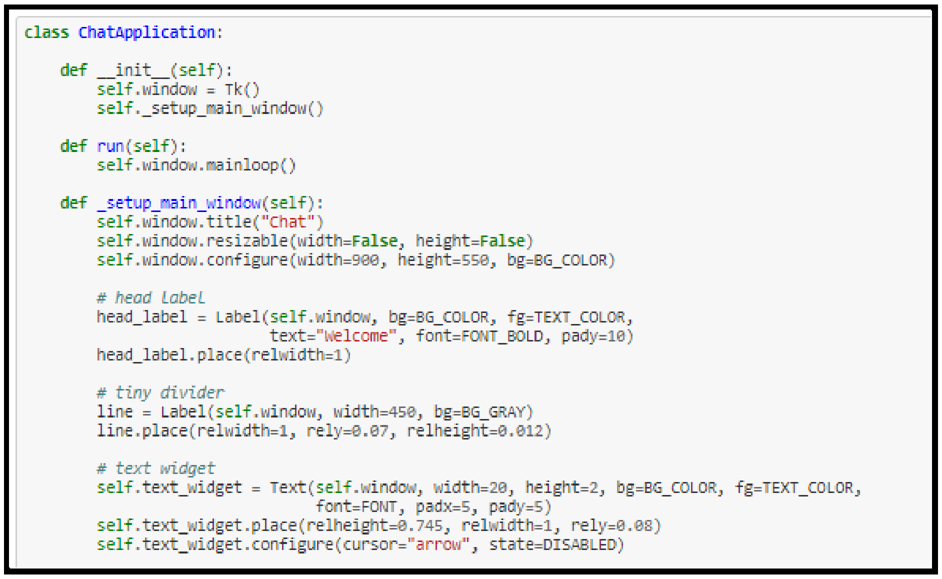
After this, we added a few cells that creates a GUI form of the model (just a more pleasant looking version of our chatbot to have a conversation with).

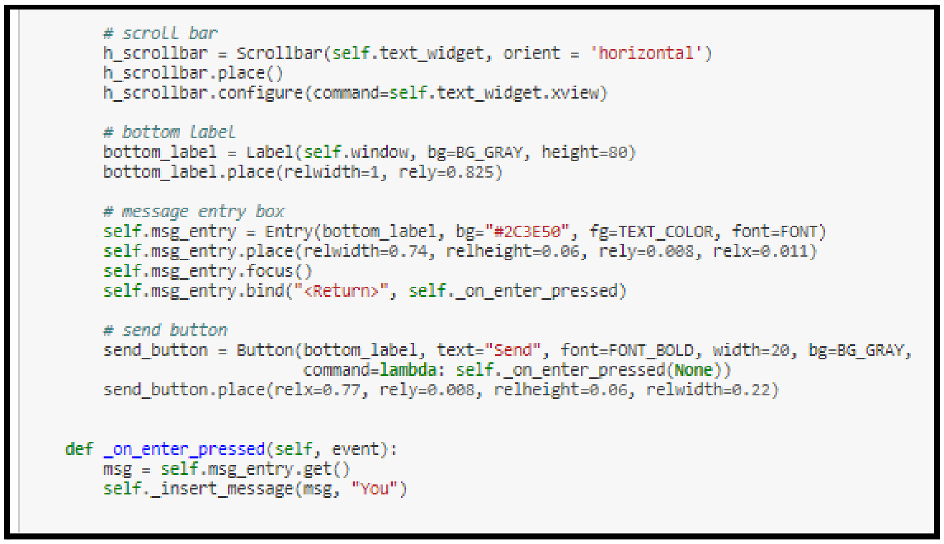


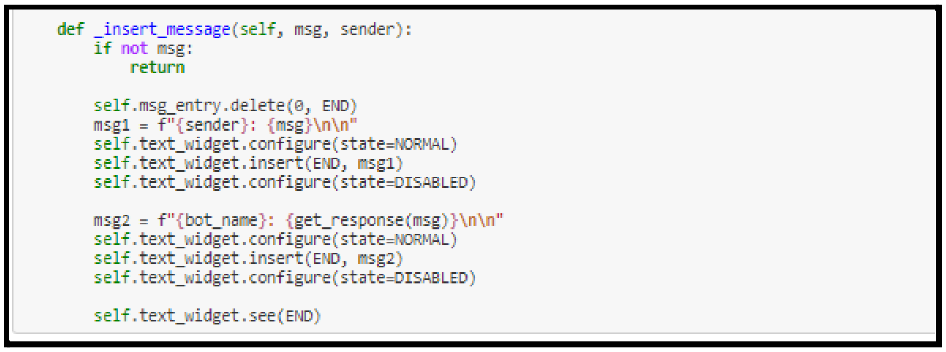
```python
def get_response(msg):

    results = model.predict(np.array([bag_of_words(msg, words)]))[0]
    results_index = np.argmax(results)
    tag = labels[results_index]

    for tg in data["intents"]:
        if tg['tag'] == tag:
            responses = tg['responses']

    return(random.choice(responses))
```



```python
BG_GRAY = "#ABB2B9"
BG_COLOR = "#17202A"
TEXT_COLOR = "#EAECEE"

FONT = "Helvetica 14"
FONT_BOLD = "Helvetica 13 bold"

bot_name = "Todd" # set the bot name for the instant message gui
```

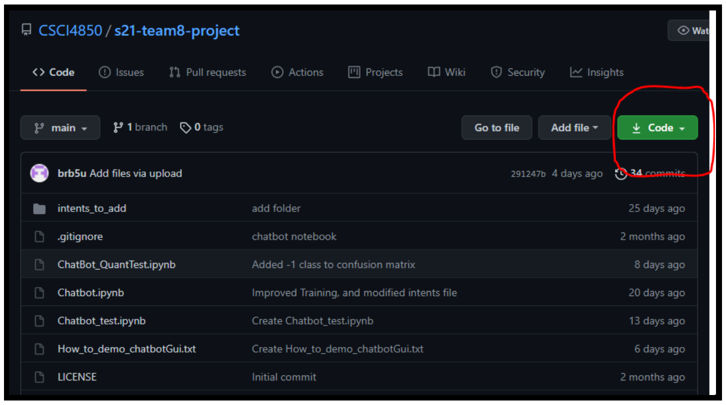

```python
class ChatApplication:

    def __init__(self):
        self.window = Tk()
        self._setup_main_window()

    def run(self):
        self.window.mainloop()

    def _setup_main_window(self):
        self.window.title("Chat")
        self.window.resizable(width=False, height=False)
        self.window.configure(width=900, height=550, bg=BG_COLOR)

        # head label
        head_label = Label(self.window, bg=BG_COLOR, fg=TEXT_COLOR,
                           text="Welcome", font=FONT_BOLD, pady=10)
        head_label.place(relwidth=1)

        # tiny divider
        line = Label(self.window, width=450, bg=BG_GRAY)
        line.place(relwidth=1, rely=0.07, relheight=0.012)

        # text widget
        self.text_widget = Text(self.window, width=20, height=2, bg=BG_COLOR, fg=TEXT_COLOR,
                                font=FONT, padx=5, pady=5)
        self.text_widget.place(relheight=0.745, relwidth=1, rely=0.08)
        self.text_widget.configure(cursor="arrow", state=DISABLED)
```


```python
        # scroll bar
        h_scrollbar = Scrollbar(self.text_widget, orient = 'horizontal')
        h_scrollbar.place()
        h_scrollbar.configure(command=self.text_widget.xview)

        # bottom label
        bottom_label = Label(self.window, bg=BG_GRAY, height=80)
        bottom_label.place(relwidth=1, rely=0.825)

        # message entry box
        self.msg_entry = Entry(bottom_label, bg="#2C3E50", fg=TEXT_COLOR, font=FONT)
        self.msg_entry.place(relwidth=0.74, relheight=0.06, rely=0.008, relx=0.011)
        self.msg_entry.focus()
        self.msg_entry.bind("<Return>", self._on_enter_pressed)

        # send button
        send_button = Button(bottom_label, text="Send", font=FONT_BOLD, width=20, bg=BG_GRAY,
                             command=lambda: self._on_enter_pressed(None))
        send_button.place(relx=0.77, rely=0.008, relheight=0.06, relwidth=0.22)


    def _on_enter_pressed(self, event):
        msg = self.msg_entry.get()
        self._insert_message(msg, "You")
```


```python
    def _insert_message(self, msg, sender):
        if not msg:
            return

        self.msg_entry.delete(0, END)
        msg1 = f"{sender}: {msg}\n\n"
        self.text_widget.configure(state=NORMAL)
        self.text_widget.insert(END, msg1)
        self.text_widget.configure(state=DISABLED)

        msg2 = f"{bot_name}: {get_response(msg)}\n\n"
        self.text_widget.configure(state=NORMAL)
        self.text_widget.insert(END, msg2)
        self.text_widget.configure(state=DISABLED)

        self.text_widget.see(END)
```

Setup for GUI Demo:

The first thing to do in order to run this demo is to download our project files form the group GitHub. Navigate to Anaconda is a resource that we found that will run the GUI for our project. First, navigate to https://github.com/CSCI4850/s21-team8-project, and click the “Code” button.



brb5u Add files via upload 291247b 4 days ago 34 commits

intents_to_add	add folder	25 days ago
.gitignore	chatbot notebook	2 months ago
ChatBot_QuantTest.ipynb	Added -1 class to confusion matrix	8 days ago
Chatbot.ipynb	Improved Training, and modified intents file	20 days ago
Chatbot_test.ipynb	Create Chatbot_test.ipynb	13 days ago
How_to_demo_chatbotGui.txt	Create How_to_demo_chatbotGui.txt	6 days ago
LICENSE	Initial commit	2 months ago

The next thing to do is to download Anaconda, which is a resource that will run the GUI for our Chatbot. Navigate to https://www.anaconda.com/**,** and follow instructions on the site to download the version of Anaconda that is compatible with your operating system. Next, open up the Anaconda program and launch the Jupyter Notebook instance. A terminal will open, followed by a webpage. Open up “Chatbot_Train_Demo_Combined.ipynb.” Run all cells, and enjoy talking to Todd the Chatbot!