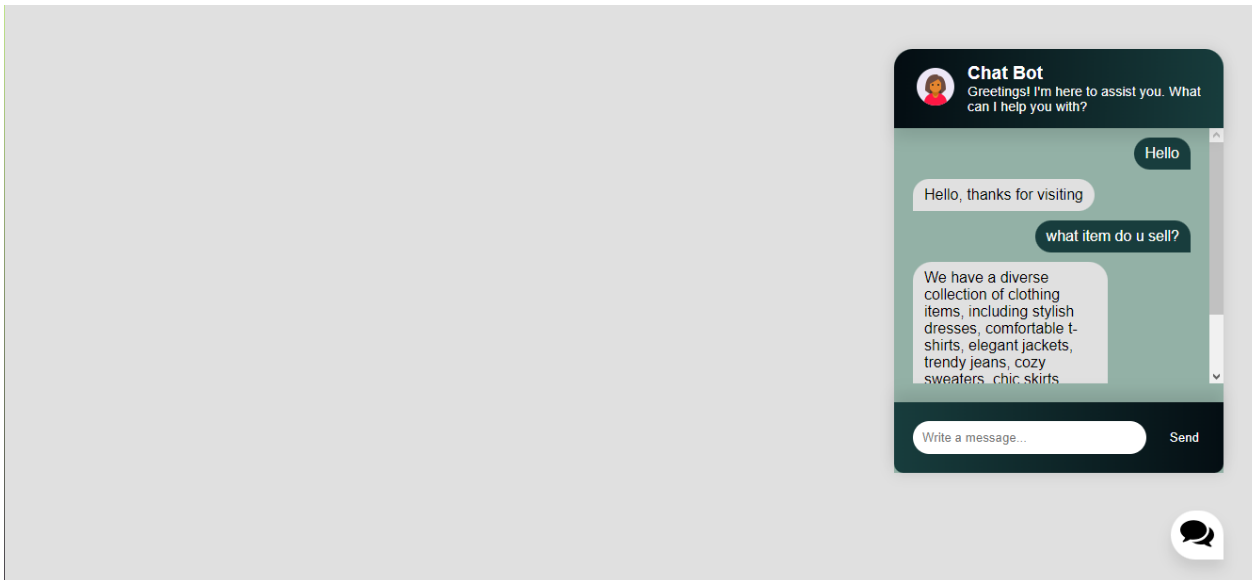
Background

Project Overview

By tackling the common problem of the absence of conversational context, this project seeks to completely transform the effectiveness of chatbots. Our main goal is to develop a custom chatbot framework that will enable a small clothing business to address client inquiries more effectively.

1. **Model Development**: Using Pytorch, we will turn conversational intent definitions into a solid model that will allow the chatbot to efficiently understand and reply to a range of client inquiries.
2. **Creation of the Framework**: We will create an advanced chatbot framework, maximizing its capacity to process user responses in an efficient manner and guaranteeing an engaging user experience.
3. **Integration of Contextual Understanding**: The last phase we'll cover is showing how the chatbot's answer processing mechanism incorporates contextual understanding, allowing it to have lively, contextually rich conversations.


Chat Bot

Greetings! I'm here to assist you. What can I help you with?

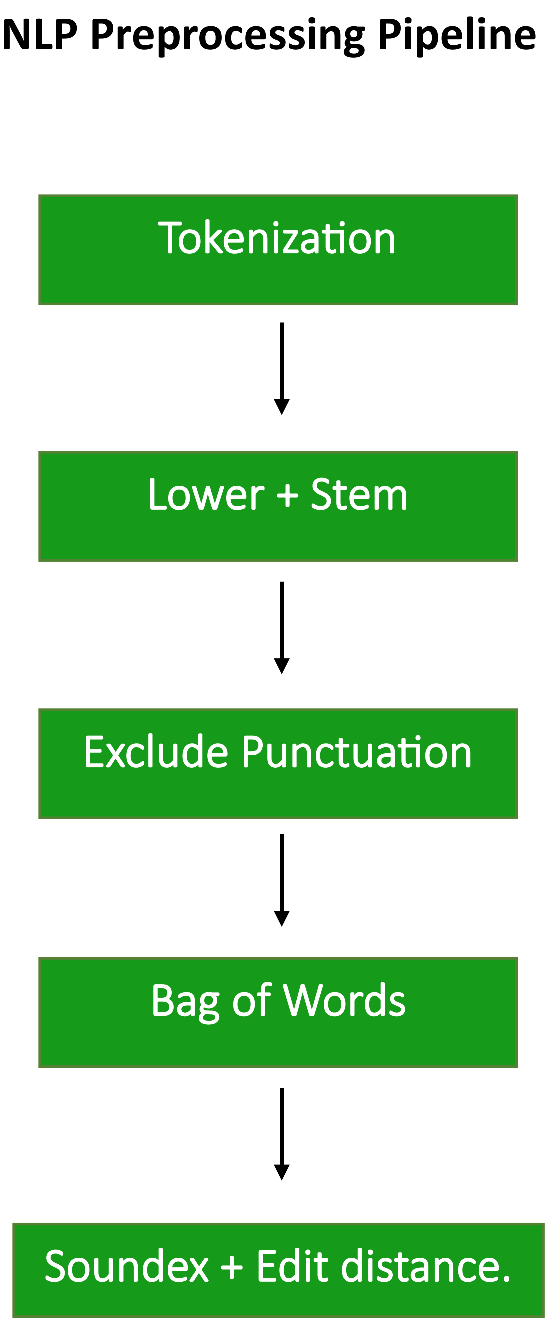
Hello

Hello, thanks for visiting

what item do u sell?

We have a diverse collection of clothing items, including stylish dresses, comfortable t-shirts, elegant jackets, trendy jeans, cozy sweaters, chic skirts

Preprocessing



intents.json

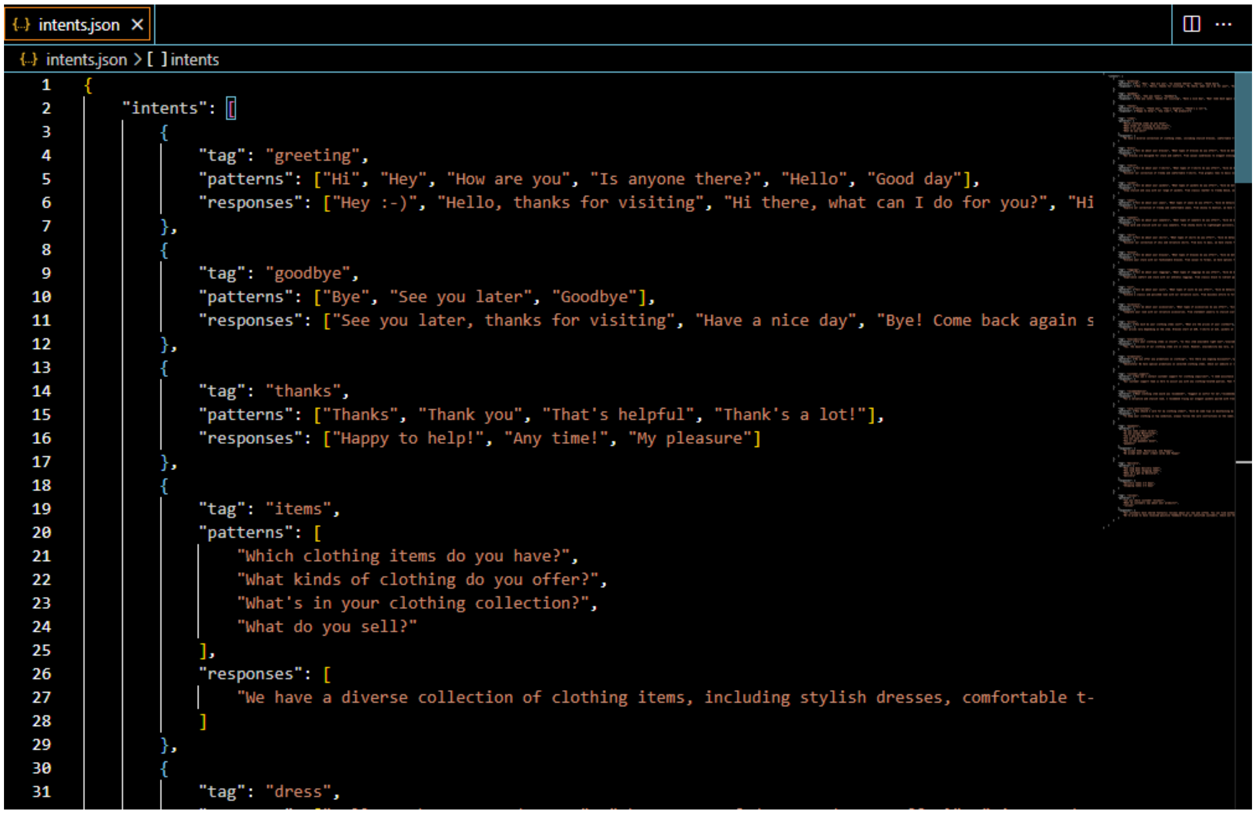
‘intents.json’ is used to define different intents and the patterns and responses that go along with them in natural language processing and chatbots that are based on neural networks. The intents.json file essentially serves as a structured data source that supplies the details required for the chatbot's natural language comprehension and response generation to be trained and run.

- **Intents:** The various categories or kinds of user intentions or inquiries that the chatbot is intended to understand and react to appropriately are represented by intents.

- **Tag:** Every intent has a special tag that acts as an identifier for that specific intent. By using these tags, the chatbot's system can better categorize user inquiries and direct them to the right response channels.

- **Patterns:** These are a list of sample words or sentences that a user could type to convey a particular idea. By using these patterns as training data, the neural network is able to learn and identify the various ways that users might phrase their queries in relation to particular intents.

- **Responses:** The pre-written messages that the chatbot can offer in response to a user's inquiry are called responses. The intended meaning and the particular context of the user's input are taken into consideration when choosing and displaying these responses.



```
{
    "intents": [
        {
            "tag": "greeting",
            "patterns": ["Hi", "Hey", "How are you", "Is anyone there?", "Hello", "Good day"],
            "responses": ["Hey :-)", "Hello, thanks for visiting", "Hi there, what can I do for you?", "Hi
        },
        {
            "tag": "goodbye",
            "patterns": ["Bye", "See you later", "Goodbye"],
            "responses": ["See you later, thanks for visiting", "Have a nice day", "Bye! Come back again s
        },
        {
            "tag": "thanks",
            "patterns": ["Thanks", "Thank you", "That's helpful", "Thank's a lot!"],
            "responses": ["Happy to help!", "Any time!", "My pleasure"]
        },
        {
            "tag": "items",
            "patterns": [
                "Which clothing items do you have?",
                "What kinds of clothing do you offer?",
                "What's in your clothing collection?",
                "What do you sell?"
            ],
            "responses": [
                "We have a diverse collection of clothing items, including stylish dresses, comfortable t-
            ]
        },
        {
            "tag": "dress",
```

Figure 1: intents.json

nltk_utils.py

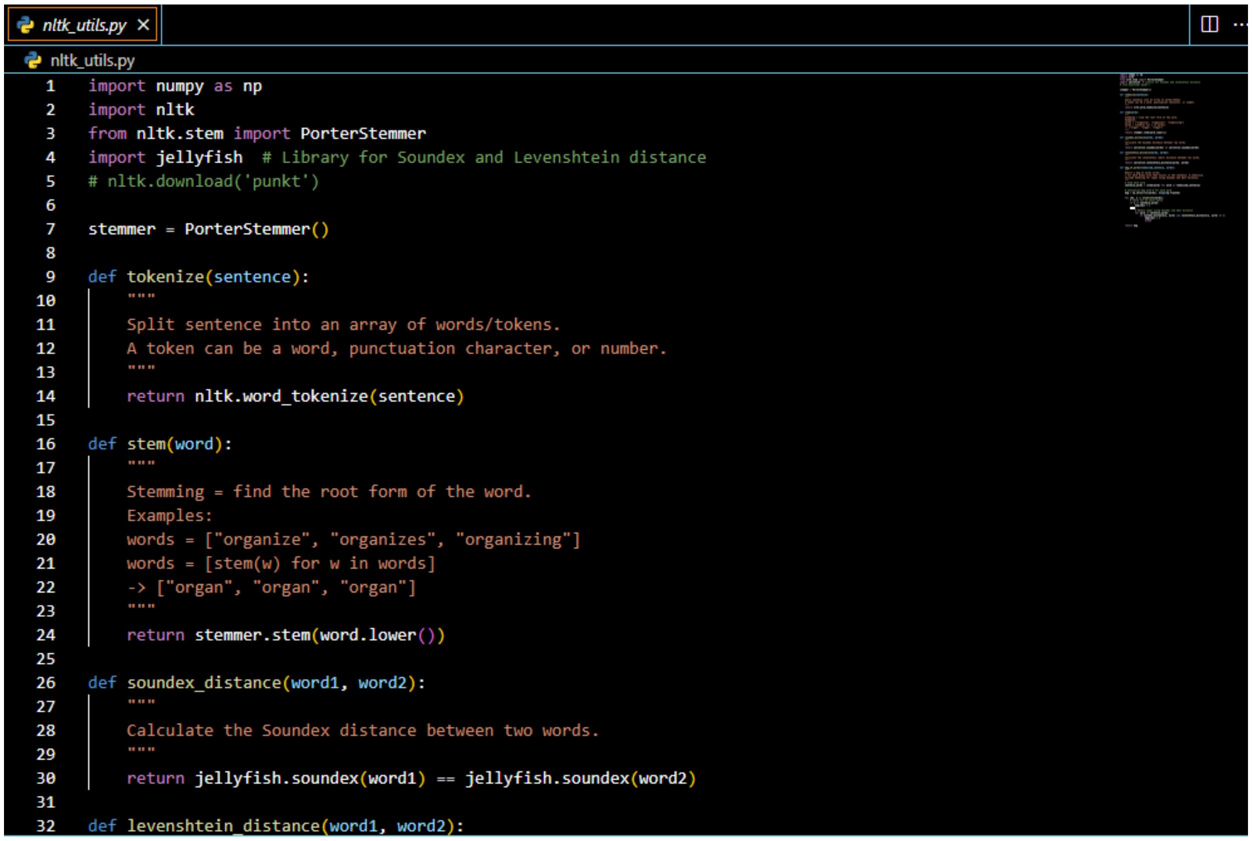
- **Tokenization:** is the process of breaking a sentence up into individual words or tokens by taking into account the words, punctuation, and numbers. It does this by utilizing the Natural Language Toolkit (nltk).

- **Stemming:** Using the stem function, stemming seeks to identify a word's root form. By reducing words to their base or root form, it makes the vocabulary simpler by allowing the system to treat words that have the same root equally.

- **Bag of Words (BoW):** For a given sentence, the bag_of_words function produces a numerical vector representation, or "bag of words". It gives a value of 1 if a word appears in the sentence and 0 otherwise, mapping each word in the sentence to a known set of words. With this method, text data is simplified without sacrificing the meaning of key words in the sentence.

- **Soundex:** In order to represent words that sound similar with the same code, Soundex is a phonetic algorithm that encodes words into a four-character code. Based on their Soundex representation, this function aids in the identification of words that sound alike.

- **Levenstein distance:** Determines how few single-character edits—insertions, deletions, or substitutions—are needed to change a word. When comparing two words based on their edit distance, this function is useful for determining how similar the words are.

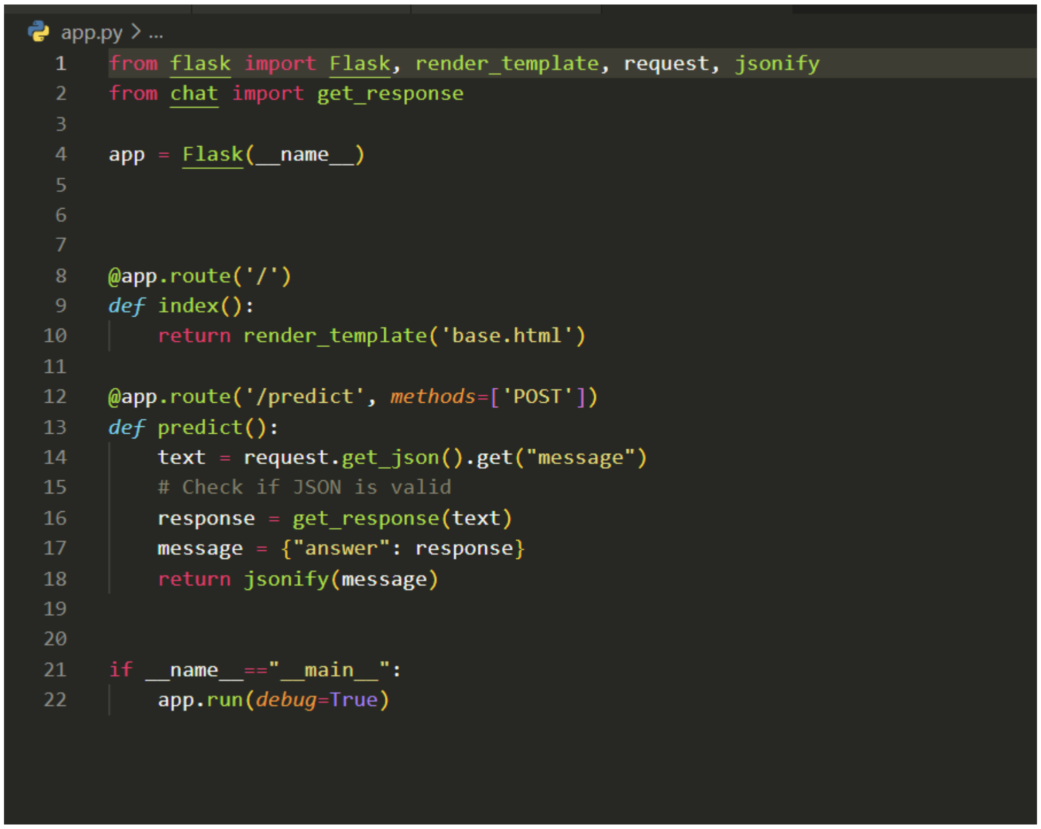


```python
import numpy as np
import nltk
from nltk.stem import PorterStemmer
import jellyfish  # Library for Soundex and Levenshtein distance
# nltk.download('punkt')

stemmer = PorterStemmer()

def tokenize(sentence):
    """
    Split sentence into an array of words/tokens.
    A token can be a word, punctuation character, or number.
    """
    return nltk.word_tokenize(sentence)

def stem(word):
    """
    Stemming = find the root form of the word.
    Examples:
    words = ["organize", "organizes", "organizing"]
    words = [stem(w) for w in words]
    -> ["organ", "organ", "organ"]
    """
    return stemmer.stem(word.lower())

def soundex_distance(word1, word2):
    """
    Calculate the Soundex distance between two words.
    """
    return jellyfish.soundex(word1) == jellyfish.soundex(word2)

def levenshtein_distance(word1, word2):
```

Figure 2: nltk_utils.py

app.py

- The Flask application is used to generate responses to user inputs via the chatbot.
- This also shows the input and output sources with index() route for the HTML template and predict() for the POST request.
- The response is created and added to a dictionary with a key value. Here the dictionary is used to convert into a JSON format.

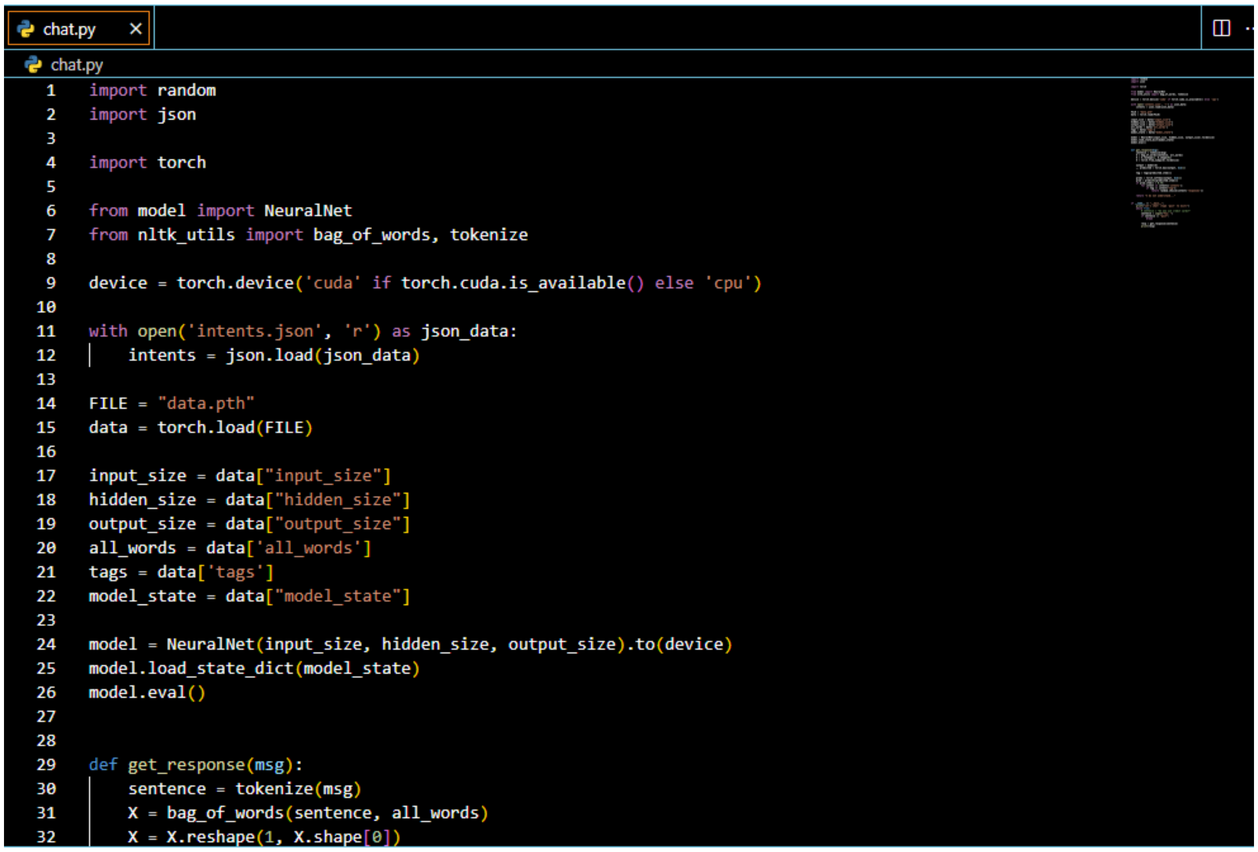


```python
from flask import Flask, render_template, request, jsonify
from chat import get_response

app = Flask(__name__)



@app.route('/')
def index():
    return render_template('base.html')

@app.route('/predict', methods=['POST'])
def predict():
    text = request.get_json().get("message")
    # Check if JSON is valid
    response = get_response(text)
    message = {"answer": response}
    return jsonify(message)


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

Figure 3:app.py

chat.py

- The chatbot uses a neural network to create an output for a given user input.
- ‘NeuralNet’ is used to train the dataset with the given intents on the dataset.
- The functions created on ‘nltk_utils.py’ is used here to tokenize and create vectors for the characters.
- A probability prediction of 75% is used to create a random response after looking the generated prediction on the intents. If a certain prediction is below 75% a response is created as ‘I do not understand’.



```python
import random
import json

import torch

from model import NeuralNet
from nltk_utils import bag_of_words, tokenize

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

with open('intents.json', 'r') as json_data:
    intents = json.load(json_data)

FILE = "data.pth"
data = torch.load(FILE)

input_size = data["input_size"]
hidden_size = data["hidden_size"]
output_size = data["output_size"]
all_words = data['all_words']
tags = data['tags']
model_state = data["model_state"]

model = NeuralNet(input_size, hidden_size, output_size).to(device)
model.load_state_dict(model_state)
model.eval()


def get_response(msg):
    sentence = tokenize(msg)
    X = bag_of_words(sentence, all_words)
    X = X.reshape(1, X.shape[0])
```

Figure 4:chat.py

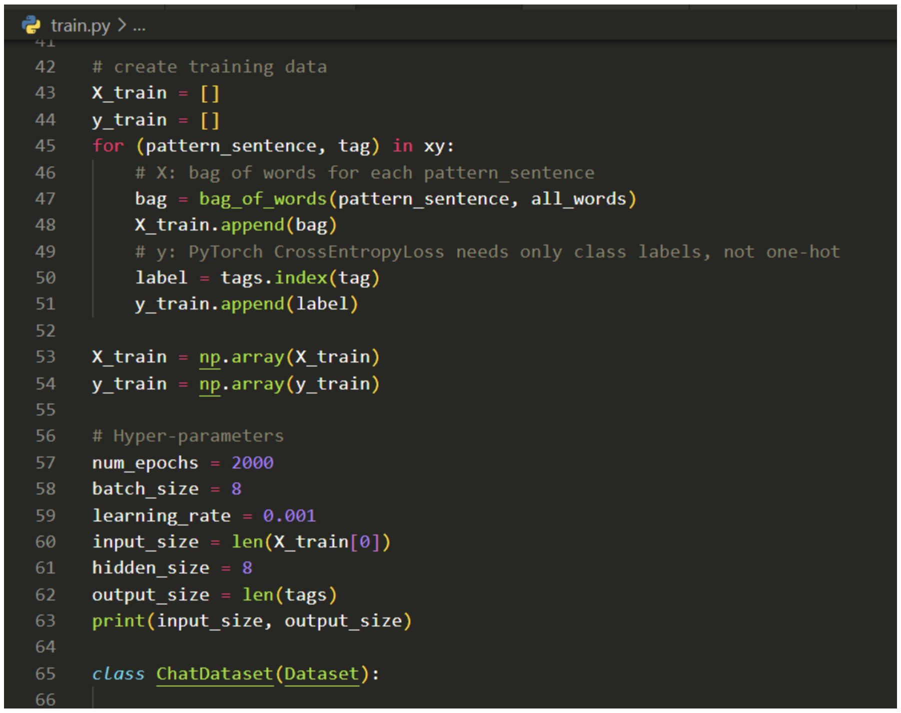
train.py

- Preparation of data
  - The 'intents. json' file, which has predefined patterns and tags for the chatbot, is where the script loads its data.
  - Tokenizing and stemming words, it preprocesses the text data and generates a bag-of-words representation for every sentence.

- Setup
  - It specifies the neural network's architecture, including the number of training epochs, batch size, and learning rate.
  - The script handles the dataset and builds a data loader for training using a customized ChatDataset class.

- Training Models
  - The script sets up the optimizer (Adam) and loss function (CrossEntropyLoss), as well as initializes the neural network model.
  - Throughout the training process, the loss is printed on a regular basis and the model is trained using the dataset.

- Saving the Model
  - When training is complete, the script stores the state dictionary of the trained model in a 'data.pth' file along with other crucial parameters.



train.py > ...

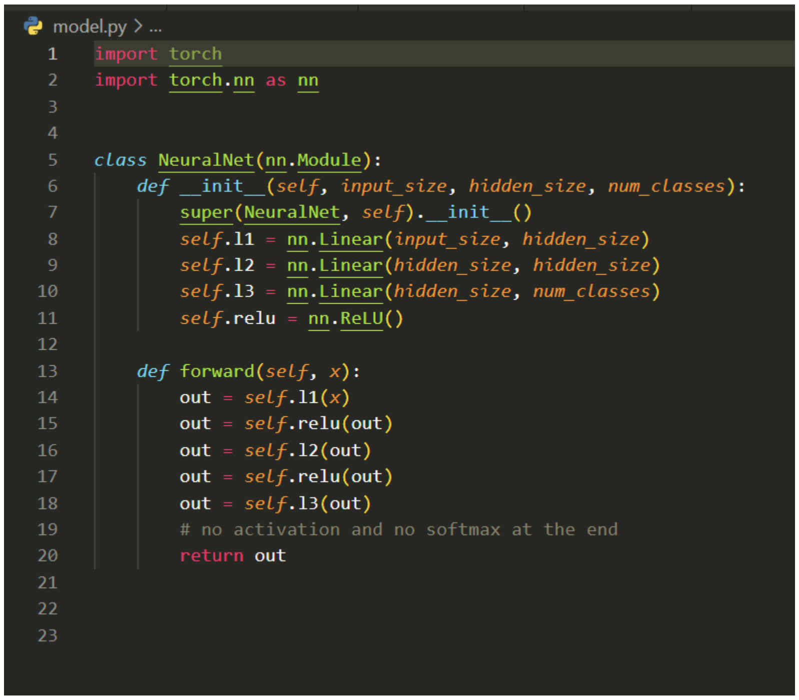
```python
# create training data
X_train = []
y_train = []
for (pattern_sentence, tag) in xy:
    # X: bag of words for each pattern_sentence
    bag = bag_of_words(pattern_sentence, all_words)
    X_train.append(bag)
    # y: PyTorch CrossEntropyLoss needs only class labels, not one-hot
    label = tags.index(tag)
    y_train.append(label)

X_train = np.array(X_train)
y_train = np.array(y_train)

# Hyper-parameters
num_epochs = 2000
batch_size = 8
learning_rate = 0.001
input_size = len(X_train[0])
hidden_size = 8
output_size = len(tags)
print(input_size, output_size)

class ChatDataset(Dataset):

```

Figure 5:train.py

model.py

- This three-layer feedforward neural network is quite basic. There are hidden_size hidden neurons and input_size input neurons in the first layer. There are hidden_size input neurons and hidden_size hidden neurons in the second layer. Neurons with hidden_size input and num_classes output make up the third layer.

- The network's three layers are traversed by an input vector x using the forward() method. A ReLU activation function is applied to each layer's output. There is no softmax function or activation when the third layer's output is returned.

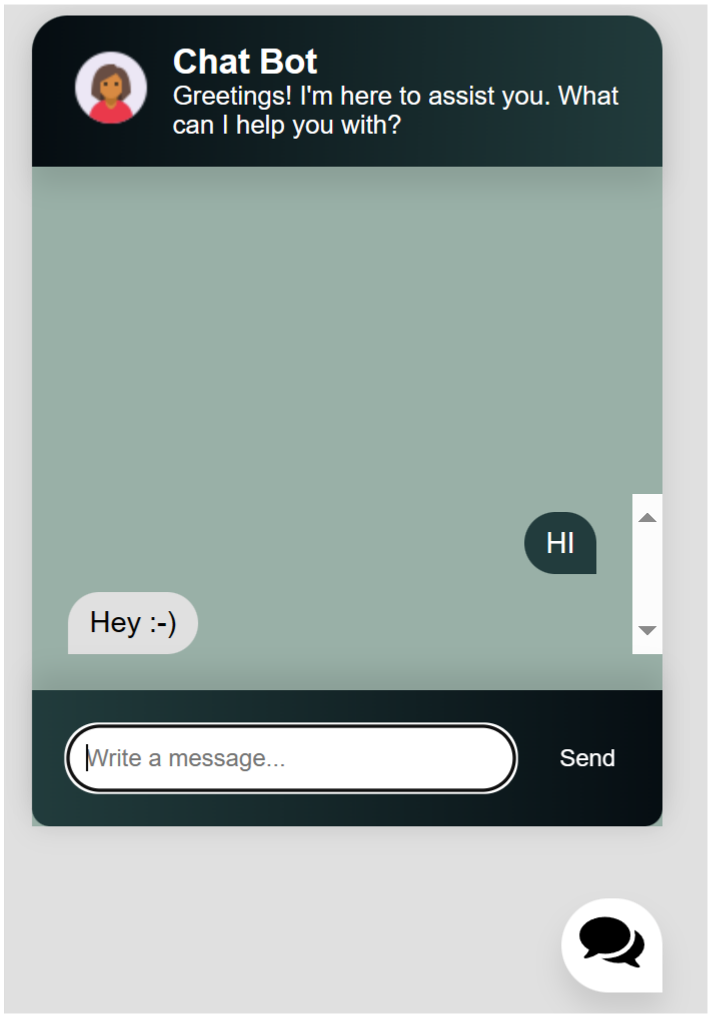


model.py > ...

```python
import torch
import torch.nn as nn


class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.l2 = nn.Linear(hidden_size, hidden_size)
        self.l3 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        # no activation and no softmax at the end
        return out
```

Figure 6:model.py

Interface


Chat Bot

Greetings! I'm here to assist you. What can I help you with?

HI

Hey :-)

Figure 7:Interface

Conclusion

The project's goal is to develop a strong model that can comprehend and respond to a variety of client inquiries regarding the pricing, products, stocks, and current offers available in the clothing business. The chatbot is prepared to provide specific and contextually appropriate conversations by highlighting the use of contextual understanding. The frequently asked questions will be answered faster than a manual operator.

References

https://www.youtube.com/watch?v=RpWeNzfSUHw&ab_channel=PatrickLoeber

https://chatbotsmagazine.com/contextual-chat-bots-with-tensorflow-4391749d0077

https://www.datacamp.com/tutorial/neural-network-models-r