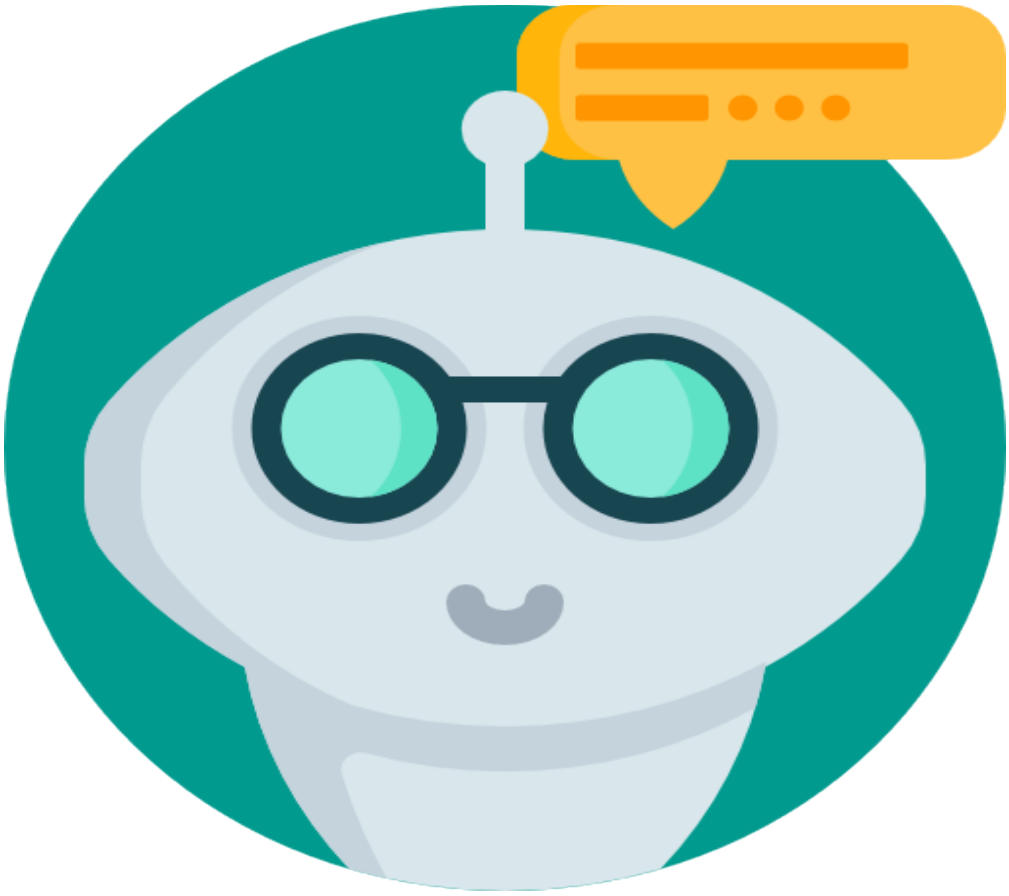
University of Management and Technology, Lahore

Documentation



Application : **Mohi's Bot**

Developer : **Ghulam-Mohi-Ud-Din**

Id : **F2021376018**

Submitted to : **Sir Mehmmod Hussain**

Aspects:

1. Python: Python is the programming language used to build the app. It provides a wide range of libraries and frameworks for web development, machine learning, natural language processing (NLP), and more.

2. Flask: Flask is a lightweight web framework in Python. It is used to create the web application part of your chatbot, handling HTTP requests and responses, routing, and rendering templates.

3. Neo4j: Neo4j is a graph database that you're using in your chatbot app. It allows you to model and store data in a graph structure, making it suitable for capturing relationships between entities and providing efficient querying capabilities.

4. AIML (Artificial Intelligence Markup Language): AIML is an XML-based language used for creating chatbot responses. It provides a way to define patterns and corresponding responses for the chatbot to generate human-like conversations.

5. Machine Learning: Machine learning is a field of study that enables computers to learn and improve from experience without being explicitly programmed. It is likely used in your chatbot to enhance its conversational capabilities and provide more accurate responses over time.

6. Prolog: Prolog is a logic programming language that allows you to define rules and facts. It is often used in chatbots to handle complex logical reasoning and inference tasks.

7. NLP (Natural Language Processing): NLP is a branch of artificial intelligence that deals with the interaction between computers and human language. It involves tasks like language understanding, sentiment analysis, named entity recognition, and more, which are used in your chatbot to interpret and respond to user input.

8. WordNet: WordNet is a lexical database that provides semantic relationships between words. It is commonly used in NLP applications to find synonyms, antonyms, hypernyms, and hyponyms, which can help improve the quality of chatbot responses.

9. Web scraping: Web scraping refers to extracting data from websites. It is likely used in your chatbot to gather information from external sources and provide relevant and up-to-date responses.

10. Social networking: Social networking integration allows your chatbot to interact with social media platforms. It enables users to perform actions such as sharing content, posting updates, or retrieving information from social networks.

These aspects collectively contribute to the functionality and intelligence of Mohi's Bot app, allowing it to understand user input, generate appropriate responses, store and retrieve data, perform logical reasoning, and integrate with external services and platforms.

System Requirements and OS :

If you want to use Mohi’s Bot for development purpose and any updation you can get it from GitHub (https://github.com/mohi0017/Chat-bot.git) and you need system atleast core i5 as there I am using Neo4j which is a heavy Graphdata base.

I designed it on my **Ubuntu**:

Distributor ID: Ubuntu
Description: Ubuntu 22.04.2 LTS
Release: 22.04
Codename: jammy

If you will use anyother OS then you can face errors and errors will arise in just Prolog comilation.

Editors:

I have used
VS-code for Flask
Sublimetext for HTML,CSS
Neo4j-Desktop (linux version 1.5.8) for neo4j database

Download:

You can download Mohi's Bot app either zip file or using clone from my GitHub account (https://github.com/mohi0017/Chat-bot.git)

Prerequisites & Installation Commands:

1. **Python**:
   - Prerequisites: None (Python can be installed on various operating systems)
   - Installation: Visit the official Python website (https://www.python.org/) and download the installer for your operating system. Follow the installation instructions provided.

2. **Flask**:
   - Prerequisites: Python (already installed)
   - Installation: Open a command prompt or terminal and run the following command:
     pip install flask

3. **Neo4j**:
   - Prerequisites: Java Development Kit (JDK) 11 or higher
   - Installation:
     - Download the Neo4j Community Edition from the official Neo4j website (https://neo4j.com/download).
- Follow the installation instructions provided for your specific operating system.

4. **AIML** (python-aiml):
- Prerequisites: Python (already installed)
- Installation: Open a command prompt or terminal and run the following command:

```
pip install python-aiml
```

5. **Machine Learning** :

- TensorFlow:
  - Prerequisites: Python (already installed)
  - Installation: Open a command prompt or terminal and run the following command:

```
pip install tensorflow
```

- Transformers (Hugging Face library):
  - Prerequisites: Python (already installed)
  - Installation: Open a command prompt or terminal and run the following command:

```
pip install transformers
```

6. **Prolog** (SWI-Prolog):
- Prerequisites: None
- Installation: Open a command prompt or terminal and run the following commands:

```
sudo apt-get update
sudo apt-get install swi-prolog
```

7. **Python development headers**:
- Prerequisites: Prolog (SWI-Prolog) (already installed)
- Installation: Open a command prompt or terminal and run the following command:

```
sudo apt-get install python3-dev
```

8. **Pyswip**:
   - Prerequisites: Prolog (SWI-Prolog) (already installed), Python development headers (already installed)
   - Installation: Open a command prompt or terminal and run the following command:
     pip install pyswip

Please note that `pyswip` requires SWI-Prolog and the Python development headers to be installed in order to function properly. By following the above steps, you should be able to install the necessary prerequisites and the `pyswip` library for Prolog integration in your Ubuntu environment.

If you encounter any issues during installation or while using `pyswip`, please refer to the library's documentation or the SWI-Prolog website for further troubleshooting and guidance.

9. **NLP** (Natural Language Toolkit - NLTK):
   - Prerequisites: Python (already installed)
   - Installation: Open a command prompt or terminal and run the following command:
     pip install nltk

10. **WordNet** (nltk.data):
   - Prerequisites: NLTK (already installed)
   - Installation: Open a Python shell and run the following commands:
     python
     import nltk
     nltk.download('wordnet')

11. **Web scraping** (Beautiful Soup):
   - Prerequisites: Python (already installed)
   - Installation: Open a command prompt or terminal and run the following command: pip install beautifulsoup4

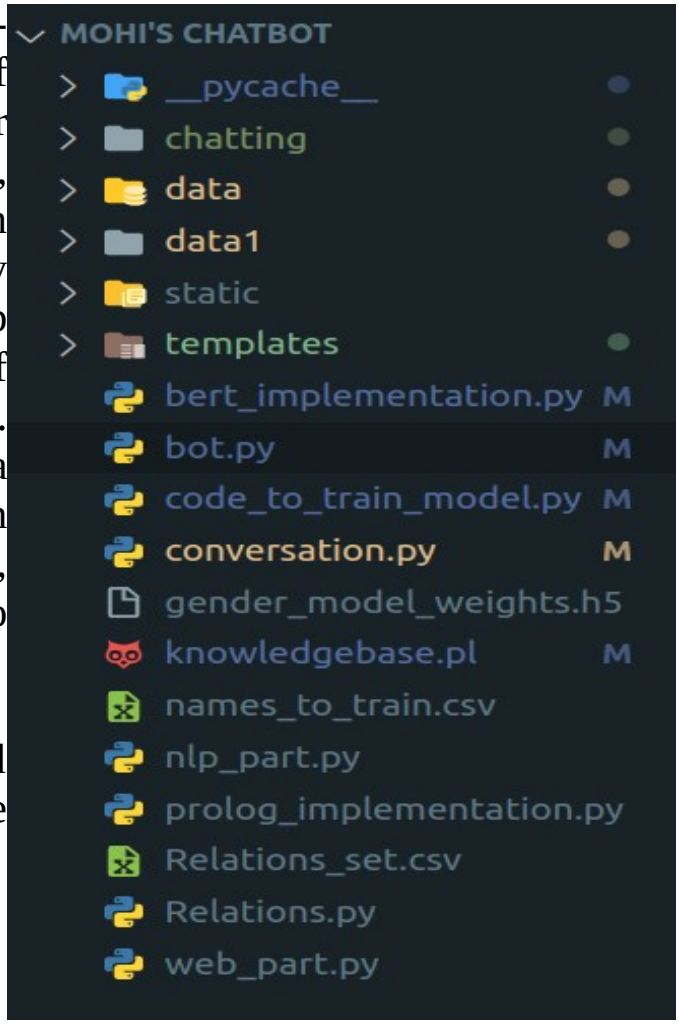
Note: Remember to use the appropriate version of pip (pip3) if you have multiple Python versions installed.

These installation commands should help you set up the necessary technologies and libraries for your Mohi's Bot app. Keep in mind that some libraries might have additional dependencies or specific installation instructions, so it's always a good idea to consult their documentation for more details.

App architecture:

In a Flask app, the basic architecture involves the use of templates and static files. Templates are responsible for generating dynamic HTML pages by incorporating variables, conditional statements, and loops, allowing for personalized and interactive content. On the other hand, static files store resources like CSS stylesheets, JavaScript files, and images that enhance the visual presentation and provide client-side interactivity. This separation of concerns promotes modular development and maintainability, enabling developers to focus on backend logic while creating visually appealing and interactive web interfaces through the combination of templates and static files. Furthermore, Application contain on a folder and that folder contains on more 2 folders (templates & static) , some python files and may also contains on some other folders.

As, I have given you my flask app architecture. When You will download and establish then your architecture will also look like that.



Explaination:

Now , I will explain to you all the aspects of folders and files

chatting folder:
This folder will use for episotic memory of chatbot.

data and data1 folder:
These folders contain on aiml files but difference between them is just of version of aiml and xml in data folder version of aiml and xml is defined in every file of aiml which restrict the use of aiml files and its patterns and the data1 folder is free of that restrictions there is no any specific version mentioned I have cleaned these files manually.

static folder:
This folder contains on css and images folders

templates folder:
This folder contains on all html templates

bert_implementation.py:
TensorFlow and Transformers libraries to perform gender classification using a pre-trained BERT (Bidirectional Encoder Representations from Transformers) model. Here's a breakdown of what the code does:

1. Imports the necessary libraries:
- `tensorflow` (imported as `tf`) for deep learning and neural network operations.
- `BertTokenizer` and `TFBertForSequenceClassification` from the `transformers` library. These classes are used for tokenization and loading the BERT model.

2. Defines the BERT model:

- The code initializes an instance of `TFBertForSequenceClassification` with the base BERT model ('bert-base-uncased') and specifies the number of labels as 2 (male and female).

3. Loads the model weights:

- The code loads the weights of a pre-trained BERT model for gender classification from the file 'gender_model_weights.h5'.

4. Initializes the BERT tokenizer:

- The code initializes a tokenizer using the 'bert-base-uncased' configuration.

5. Defines a function to predict gender:

- The `predict_gender` function takes a name as input.

- It encodes the name using the BERT tokenizer, applying truncation and padding.

- Creates a TensorFlow dataset from the encoded input data.

- Uses the loaded BERT model to make predictions on the input dataset.

- Determines the predicted label (0 for male, 1 for female) by finding the index of the maximum value in the logits.

- Returns the predicted gender as a string ('male' or 'female').

Please note that to run this code successfully, you'll need to obtain the 'gender_model_weights.h5' file, which is about 420MB in size. The code also mentions an alternative approach where you can train your own gender classification model using the provided dataset ('names_to_train.csv') and the 'code_to_train_model.py' file, which uses the BERT classifier.

bot.py:

it is a Flask web application that serves as a chatbot called "Mohi's Bot." The application integrates various modules and libraries to handle user interactions and provide responses based on predefined rules and patterns.

Let's break down the code and explain each concept:

1. Importing Libraries:

- `os`: Provides a way to interact with the operating system.
- `Flask`: A web framework for building the application.
- `render_template`: Renders HTML templates for the webpages.
- `request`: Handles HTTP requests.
- `redirect`: Redirects the user to a different URL.
- `jsonify`: Converts Python objects to JSON format.
- `Graph` (from `py2neo`): Interacts with Neo4j graph database.
- `Kernel` (from `aiml`): Implements the AIML (Artificial Intelligence Markup Language) chatbot kernel.
- `datetime`, `date`: Manipulates date and time values.

2. Initializing Flask:
- `app = Flask("Mohi's Bot", template_folder='templates')`: Creates a Flask application with the name "Mohi's Bot" and sets the template folder.

3. Database and Global Variables:
- `graph`: Initializes a connection to a Neo4j graph database.
- `check_siginday`, `total_days`, `user`: Global variables used to track user login information.

4. Route Definitions:
- `@app.route('/')`, `@app.route('/getlogin')`, `@app.route('/registration')`, `@app.route("/home")`: Define the URL routes and associate them with corresponding functions that handle the requests.
- For example, when a user accesses the root URL (`/`), the `login()` function is executed and renders the "mohi's_home.html" template.

5. User Registration and Login:
- `/signup`, `/login`: These routes handle user registration and login processes using the HTTP methods POST.
- The user's information (username, email, password) is collected from the HTML forms and stored in the Neo4j graph database.

6. AIML Chatbot Initialization:
- `my_bot = Kernel()`: Creates an instance of the AIML Kernel, which is responsible for processing AIML files and generating responses.

7. AIML Files Loading:
- `load_aiml_files()`: Loads AIML files from the "data1" directory and teaches the chatbot using `my_bot.learn()`.

8. Chatbot Response Generation:
- `chat_bot_reply(message)`: Takes a user message as input, passes it to the AIML chatbot kernel (`my_bot.respond()`), and returns the generated response.

9. Data Preprocessing and Natural Language Processing (NLP):
- Several functions from the `nlp_part` module are called to perform text processing, such as auto-spelling correction, sentence splitting, part-of-speech tagging, named entity recognition, etc.

10. Web Scraping and WordNet:
- The chatbot checks if the response from the AIML chatbot kernel is empty or contains specific phrases. In such cases, it performs web scraping (`get_Query()`) and retrieves definitions from WordNet to enhance the response.

11. Route for Chatbot Interaction:
- `/get`: This route handles user interactions with the chatbot. The user's message is passed to `chat_bot_reply()` for generating a response.
- The response, along with other values, is returned as a JSON object.

12. File Logging:
- The code includes functionality to log user interactions with the chatbot in separate text files (`chatting/episode_{total_days}.txt`).
- The user

's query and the bot's response are appended to the file.

13. Application Execution:

- `if __name__ == "__main__":`: Ensures that the application is only executed if the script is run directly, not imported as a module.

- `app.run(debug=True, host='0.0.0.0', port='8000')`: Runs the Flask application on the local development server with debugging enabled.

Overall, the code sets up a Flask web application that integrates an AIML chatbot and interacts with a Neo4j graph database to handle user registration, login, and provide responses based on predefined patterns. The application also incorporates NLP techniques, web scraping, and logging of user interactions.

code_to_train_model.py:

This code demonstrates the fine-tuning of a pre-trained BERT model for gender classification using TensorFlow 2.x and the Transformers library. It performs a similar task as the previous code, but with some changes specific to TensorFlow. Let's break it down step-by-step:

1. Importing Libraries:

- `tensorflow`: The main library for building and training deep learning models in TensorFlow.

- `BertTokenizer`, `TFBertForSequenceClassification`: Classes from the Transformers library specific to BERT model usage with TensorFlow.

- `pandas`: A library for data manipulation and analysis.

- `train_test_split`: A function from scikit-learn to split the dataset into training and validation sets.

- `numpy`: A library for numerical computing.

2. Loading the Dataset:

- Reads the dataset from a CSV file named "names_to_train.csv" using pandas and assigns the first column to the `names` variable and the second column to the `Labels` variable.

3. Preparing the Data:

- Converts the `names` and `labels` columns to Python lists for further processing.

4. Converting Labels to Numerical Values:

- Defines a dictionary, `label_to_int`, that maps the labels "male" and "female" to numerical values 0 and 1, respectively.

- Converts the original string labels to numerical values based on the `label_to_int` mapping.

5. Splitting the Dataset:

- Splits the dataset into training and validation sets using `train_test_split` function from scikit-learn. It assigns 80% of the data to the training set and 20% to the validation set.

6. Loading the Pre-trained BERT Model and Tokenizer:

- Initializes the BERT tokenizer and model using the `BertTokenizer.from_pretrained()` and `TFBertForSequenceClassification.from_pretrained()` methods, respectively. The model is loaded with the "bert-base-uncased" configuration, which is a pre-trained BERT model with uncased (lowercase) text.

7. Tokenizing the Input Names:

- Tokenizes the input names for both the training and validation sets using the tokenizer's `tokenizer()` method and assigns the encoded inputs to `train_encodings` and `val_encodings`, respectively.

- Converts the encodings into NumPy arrays with appropriate data types.

8. Creating TensorFlow Datasets:

- Converts the data into TensorFlow datasets using `tf.data.Dataset.from_tensor_slices()`.

- Batches the data with a batch size of 16.

9. Fine-tuning the BERT Model:

- Sets up the optimizer (`Adam` with a learning rate of 1e-5), loss function (`SparseCategoricalCrossentropy`), and accuracy metric (`SparseCategoricalAccuracy`).

- Compiles the model with the defined optimizer, loss, and metric.

- Trains the model on the training data for 5 epochs using the `fit()` method with the validation data for validation.

10. Saving the Trained Model Weights:
- Saves the trained model weights using the `save_weights()` method.

11. Loading the Trained Model:
- Loads the trained model using the `TFBertForSequenceClassification.from_pretrained()` method and loads the saved model weights using the `load_weights()` method.

12. Predicting Gender based on a Name:
- Defines a function, `predict_gender(name)`, to predict the gender based on a given name.
- Tokenizes the input name using the tokenizer.
- Creates a TensorFlow dataset from the tokenized input.
- Makes predictions using the trained model and retrieves the predicted label.
- Maps the predicted label to the corresponding gender ("male

" or "female").

13. Getting Gender from User Input:
- Prompts the user to enter a name.
- Calls the `predict_gender()` function to predict the gender based on the input name.
- Prints the predicted gender for the given name.

Note: The code assumes that you have access to a GPU (CUDA) for faster training and inference. If not, it falls back to CPU execution.

conversation.py:
It's code demonstrates the implementation of a console-based chatbot using the AIML (Artificial Intelligence Markup Language) library. AIML is a widely-used markup language for creating chatbots.

Let's go through the code step-by-step:

1. Importing Libraries:
- `Kernel` from `aiml`: The main class from the AIML library for creating an AIML kernel, which is the brain of the chatbot.
- `os`: A module for interacting with the operating system.

2. Creating the AIML Kernel and Loading AIML Files:
- Initializes an AIML kernel object using `Kernel()`.
- Defines a function, `load_aiml_files()`, to load AIML files from a directory.
- The `aiml_directory` variable holds the path to the directory containing AIML files.
- Retrieves a list of AIML files using `os.listdir()` and filters for files with a ".aiml" extension.
- Iterates over the AIML files and calls the `learn()` method of the AIML kernel to load the AIML file.

3. Defining the Chat Bot Reply Function:
- Creates a function, `chat_bot_reply(message)`, to get a response from the chatbot based on a given message.
- The function calls the `respond()` method of the AIML kernel to get a response.
- If the response is "unknown," it sets the response to `None`.
- Handles any exceptions that occur during the response generation and sets the response to `None` in case of an error.
- Returns the response.

4. Loading AIML Files:
- Calls the `load_aiml_files()` function to load the AIML files into the AIML kernel.

5. Chatting with the Bot:
- Enters an infinite loop to continuously interact with the chatbot.
- Prompts the user to enter a query using the `input()` function.

- Calls the `chat_bot_reply()` function to get a response from the chatbot based on the user's query.
- If a response is obtained, it prints "Bot >" followed by the response.
- If no response is obtained, it prints "Bot > :)" to indicate a neutral response.

The code allows users to have a conversation with the chatbot by entering queries, and the chatbot responds based on the AIML files it has learned. If the chatbot encounters an unknown query, it responds with a neutral expression ":)".

gender_model_weights.h5:
It is trained model to predict gender from a name.It's size is about 420Mbs you can get it from my google drive
https://drive.google.com/file/d/1W0qq_wzSO30qMj2Re1gTtB5PdlEWXZTG/view?usp=sharing

knowledgebase.pl:
It is a Prolog program that defines various relationship predicates and a list of individuals with their genders. The purpose of the code is to establish relationships between individuals based on their gender and parent-child connections.

Let's go through the code section by section:

1. Discontiguous Predicates:
- This section contains multiple directives starting with `:- discontiguous`. These directives inform the Prolog interpreter that the predicates defined subsequently may not be defined contiguously in the file. This allows for more flexibility in defining the predicates.

2. Individual Definitions:
- This section lists the individuals and their genders. For example, it defines males like 'john', 'jim', 'tom', and 'sam', and females like 'mary', 'ann', 'lisa', and 'sara'.

3. Parent-Child Relationships:

- This section establishes parent-child relationships between the individuals defined earlier. For example, it defines that 'john' is the parent of 'jim', 'ann', 'lisa', and 'tom'.

4. Relationship Predicates:

- This section defines various relationship predicates based on the established parent-child relationships and the genders of the individuals.

- For example, `husband(X, Y)` defines a husband relationship where `X` is the husband and `Y` is the wife. It checks if there exists a common child `Z` between `X` and `Y`.

- Similarly, other predicates like `wife`, `spouse`, `sister`, `married`, `brother`, `father`, `mother`, `child`, `grandparent`, `grandchild`, `sibling`, `son`, `daughter`, `grandfather`, `grandmother`, `grandson`, `granddaughter`, `uncle`, `aunt`, `nephew`, `niece`, `cousin`, `half_sibling`, `step_sibling`, `ancestor`, `descendant`, `sister_in_law`, `brother_in_law`, `father_in_law`, `mother_in_law`, `nephew_in_law`, `niece_in_law`, `cousin_in_law`, `related` are defined with their respective relationship criteria.

5. Example Facts:

- The code also includes additional facts at the end, such as `male('Good')` and `male('Lahore')`, which define two additional males.

The purpose of this Prolog program is to define a knowledge base of individuals and their relationships. By using these defined predicates, you can query the knowledge base to determine various relationships between the individuals, such as finding the parents of a person, checking if two individuals are siblings, or determining the cousins of a person.

The code provides a foundation for querying relationships between individuals in a Prolog environment. Additional facts and rules can be added to further expand the knowledge base and allow for more complex relationship queries.

names_to_train.csv:
This is a csv file containing names dataset of every religion. Number of names is about to 96917.

nlp_part.py:
It's code includes several functions and imports used for natural language processing and text analysis. Let's go through each part:

1. Import Statements:
- `random`: A module for generating random numbers and making random choices.
- `WordNetLemmatizer` from `nltk.stem`: A lemmatization tool used for reducing words to their base or root form.
- `CountVectorizer` from `sklearn.feature_extraction.text`: A tool for converting text documents into a matrix of token counts.
- `LatentDirichletAllocation` from `sklearn.decomposition`: A topic modeling technique based on the Dirichlet distribution.
- `socket`: A module providing access to various networking interfaces, including IP address retrieval.
- `spacy`: An open-source library for natural language processing.
- `SentimentIntensityAnalyzer` from `nltk.sentiment`: A tool for sentiment analysis using a pre-trained model.
- `ne_chunk`, `pos_tag`, `word_tokenize`, `sent_tokenize` from `nltk`: Functions and classes for named entity recognition, part-of-speech tagging, and tokenization.
- `Tree` from `nltk.tree`: A class representing a hierarchical structure for syntactic parsing.
- `Relations`: A custom module or class for managing relationships between entities.
- `stopwords`, `wordnet`, `opinion_lexicon` from `nltk.corpus`: Resources for stop words, WordNet lexical database, and opinion lexicon.

2. `replace_withname(query, name)`: A function that replaces personal pronouns (e.g., 'I', 'me', 'my', 'mine') in a given query with a specified name. It tokenizes the query, checks for pronouns, and replaces them accordingly.

3. `NER(text, user)`: A function for named entity recognition (NER) that extracts named entities and their relationships from the text. It tokenizes the text, performs NER using the `ne_chunk` function, identifies relevant entities and relationships, and creates or updates relations using the custom `Relations` module.

4. `autospell(text)`: A function that corrects spelling errors in a given text using the GingerIt library. It also restores proper casing using the truecase library. The function returns the corrected text.

5. `word_synonyms(word)`: A function that retrieves synonyms of a given word using WordNet. It iterates through the synsets of the word and its lemmas, collecting the lemma names as synonyms.

6. `sentence(query)`: A function that tokenizes a query into sentences using `sent_tokenize` from NLTK. It returns a list of sentences.

7. `print_random_string(strings)`: A function that randomly selects and returns a string from a given list of strings.

8. `chart(query)`: A function that performs sentiment analysis on a given query using the SentimentIntensityAnalyzer from NLTK. It calculates sentiment scores (positive, negative, neutral) using a pre-trained model and returns them as a list.

9. `analyze_sentiment(text)`: A function that analyzes the sentiment of a given text using the SentimentIntensityAnalyzer. It calculates the sentiment score and returns 'Positive', 'Negative', or 'Neutral' based on the score threshold.

10. `detect_emotion(text)`: A function that detects the dominant emotion in a given text. It tokenizes the text, removes stop words, and assigns scores to different emotions based on the presence of positive and negative words or synonyms. It returns the dominant emotion (e.g., 'joy', 'sadness', 'anger', 'fear').

11. `extract_topics(texts)`: A function that extracts topics from

a given text. It tokenizes the text into sentences, performs lemmatization, removes stop words, and creates a document-term matrix. It applies Latent Dirichlet Allocation (LDA) to identify topics and randomly selects words from each topic. It combines extracted topics with named entities using named entity recognition. The function returns a list of topics or topic-entity pairs.

12. `getIpAdrress()`: A function that retrieves the IP address of the local machine using a socket connection. It connects to the Google DNS server and extracts the IP address from the socket.

13. `get_definition(query)`: A function that retrieves the definition of a word using WordNet. It finds synonyms of the word and retrieves the definition from the second synset. It returns the definition along with a randomly selected synonym.

These functions and imports cover a range of text analysis tasks, including entity recognition, sentiment analysis, spelling correction, synonym retrieval, topic extraction, and IP address retrieval.

prolog_implementation.py:

The code is using the `pyswip` library, which is a Python interface to the SWI-Prolog system. It allows you to interact with Prolog logic programming from Python. Let's break down the code and explain each aspect:

1. Import Statement:

- `from pyswip import Prolog`: This imports the `Prolog` class from the `pyswip` library, enabling interaction with Prolog.

2. Prolog Initialization:

- `prolog = Prolog()`: This creates an instance of the `Prolog` class, which represents the Prolog interpreter.

3. Knowledge Base Consultation:

- `prolog.consult('knowledgebase.pl')`: This instructs the Prolog interpreter to consult the Prolog file 'knowledgebase.pl'. It loads the facts and rules defined in the file into the Prolog knowledge base.

4. Function: `write_fact_and_rules(query)`

- This function appends a Prolog query to the 'knowledgebase.pl' file. It allows you to dynamically add new facts and rules to the knowledge base.

5. Function: `get_results(query)`

- This function takes a Prolog query as input and returns the results of the query as a list.

- It uses the `query()` method of the `Prolog` instance to execute the query and convert the results into a Python list.

6. Function: `print_results(results)`

- This function takes the results of a Prolog query as input and prints them in a readable format.

- It iterates over each result and prints the key-value pairs.

7. Function: `check_relation(name, relation)`

- This function checks the relation between a given name and a specific relation using Prolog.

- It constructs a Prolog query by concatenating the relation, 'X' (representing the person with the relation), and the given name.

- It calls the `get_results()` function to execute the query and retrieve the results.

- If there are results, it prints the persons who have the specified relation with the given name using the `print_results()` function.

- If there are no results, it prints a message indicating that no one has the specified relation with the given name.

Overall, the code provides a way to interact with a Prolog knowledge base by writing facts and rules to a file, querying the knowledge base, and retrieving and printing the results. It allows you to check relations between individuals based on the defined rules and facts in the knowledge base.

Relations_set.csv:

This file contains on 540 relations which can exists between any two or more objects . It is use to identify relation in text which Mohi's Bot get from user. When relation will identify successfully then it go for neo4j , prolog , e.t.c.

Relations.py:

The code demonstrates the implementation of a system that creates relationships and nodes in a graph database (Neo4j) based on data stored in a CSV file. Let's go through the code and explain it in detail:

1. Import Statements:

- `from py2neo import Graph`: This imports the `Graph` class from the `py2neo` library, which provides functionality to interact with the Neo4j graph database.

- `import pandas as pd`: This imports the `pandas` library, which is used for data manipulation and analysis.

- `from prolog_implementation import write_fact_and_rules`: This imports a custom module `prolog_implementation` that contains the `write_fact_and_rules` function.

2. Graph Initialization:

- `graph = Graph("bolt://localhost:7689", auth=("neo4j", "12345678"))`: This creates an instance of the `Graph` class, connecting to the Neo4j database running on `localhost` with the given authentication credentials.

3. CSV Data Loading:

- `data = pd.read_csv("Relations_set.csv", header=None)`: This reads the CSV file named "Relations_set.csv" using `pandas` and stores the data in a DataFrame.

- `df = pd.DataFrame(data)`: This converts the DataFrame into a `pandas` DataFrame object for easier data handling.

4. Function: `relationships()`

- This function retrieves a list of relationships from the first column of the DataFrame (`df`).

5. Global Variable: `relations_list`

- This variable stores the list of relationships obtained from the `relationships()` function.

6. Function: `create_relation(Person_names_rel)`

- This function creates relationships between nodes in the Neo4j graph database based on a given list of person names and relationships (`Person_names_rel`) and it's length is 3.

- It iterates over each name in the list.

- If the name is a recognized relationship (present in `relations_list`), the function performs the following steps:

- It generates a Prolog rule by combining the current relationship, the previous name in the list, and the next name in the list.

- It calls the `write_fact_and_rules()` function (from the `prolog_implementation` module) to write the Prolog rule to a knowledge base file.

- It checks if the previous and next names are not already in the list of relationships.

- If not, it executes a Cypher query to create a relationship between the previous and next names in the Neo4j graph database.

- The function handles different scenarios based on the position of the name in the list (first, last, or in between).

7. Function: `create_Node(name, gender)`

- This function creates a new node in the Neo4j graph database with the given `name` and `gender`.

- It executes a Cypher query using the `graph.run()` method to create a node with the specified properties (name and gender).

- It also generates a Prolog fact based on the gender and name, and writes it to the knowledge base file using the `write_fact_and_rules()` function.

The code combines Neo4j database operations and Prolog logic programming to create relationships and nodes based on data from a CSV file. It leverages the `py2neo` library for Neo4j interaction and `pandas` for data handling, while the `prolog_implementation` module aids in writing Prolog rules and facts to a knowledge base file.

web_part.py:

The code you provided includes a set of functions that allow you to search for a query on Wikipedia, retrieve the page content, clean the HTML, and shorten the text to a specified number of sentences. Let's go through each function and explain them in detail:

1. Import Statements:

- `import re`: This imports the regular expression module, which is used for pattern matching and text manipulation.

- `import requests`: This imports the `requests` library, which is used to send HTTP requests to the Wikipedia API and retrieve data.

- `from bs4 import BeautifulSoup`: This imports the `BeautifulSoup` class from the `bs4` module, which is used for HTML parsing.

2. Function: `search_wikipedia(query)`

- This function searches for a given `query` on Wikipedia using the Wikipedia API.

- It constructs the URL and sets the necessary parameters for the API request.

- It sends the request using `requests.get()` and checks if the response status code is 200 (indicating a successful request).

- If the response is successful, it extracts the search results from the JSON response and retrieves the page ID of the first search result.

- It then calls the `get_wikipedia_page()` function to retrieve the content of the Wikipedia page with the obtained page ID.

- If no search results are found or there is an error in the process, it returns `None`.

3. Function: `get_wikipedia_page(page_id, num_sentences=3)`

- This function retrieves the content of a Wikipedia page based on its `page_id`.

- It constructs the URL and sets the necessary parameters for the API request.

- It sends the request using `requests.get()` and checks if the response status code is 200.

- If the response is successful, it extracts the page content from the JSON response and cleans the HTML using the `clean_html()` function.
- It then shortens the cleaned text to a specified number of sentences using the `shorten_text()` function.
- The shortened text is returned, or `None` if there is an error.
4. Function: `clean_html(html)`
- This function takes an HTML string as input and cleans it by removing unwanted elements.
- It uses `BeautifulSoup` to parse the HTML and remove specific elements such as superscripts, tables, and styles.
- The cleaned text is obtained using the `get_text()` method of `BeautifulSoup`.
- The function returns the cleaned text.

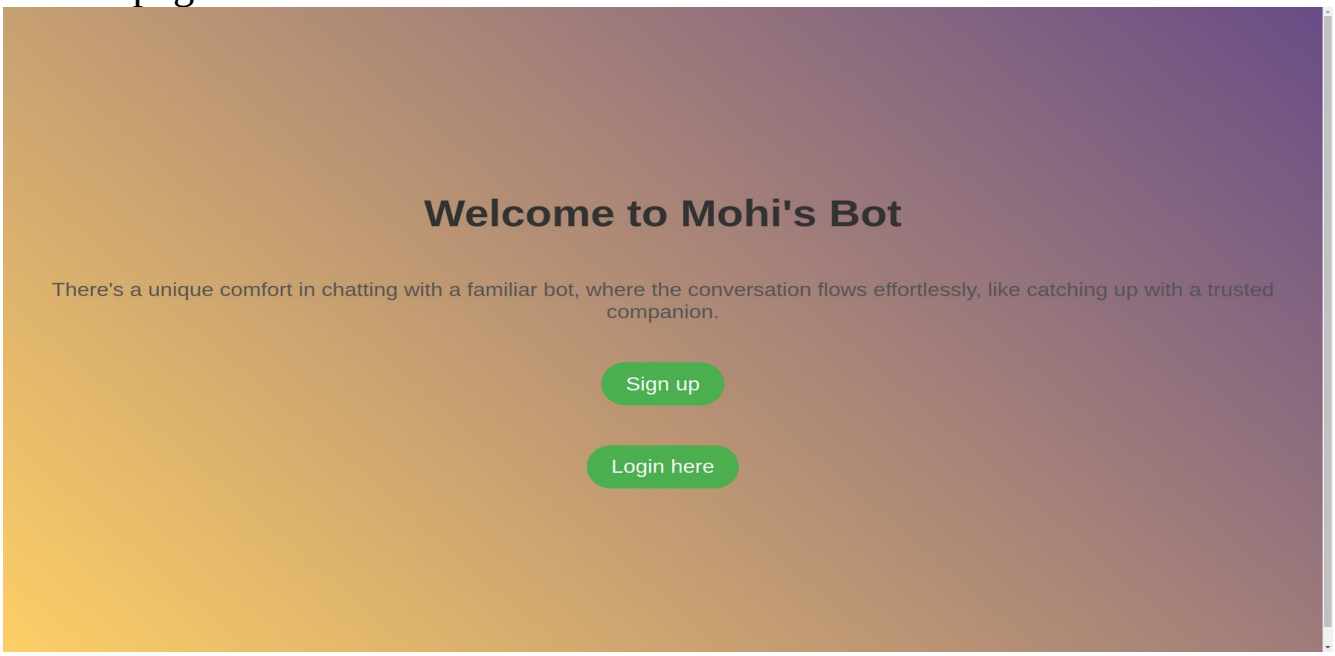
5. Function: `shorten_text(text, num_sentences)`
- This function takes a text string and a number of sentences as input and shortens the text to the specified number of sentences.
- It splits the text into paragraphs based on double newline characters (`\n\n`).
- If paragraphs are found, it concatenates them into a single text string and removes citations and references using regular expressions (`re.sub()`).
- It splits the text into sentences using a regular expression pattern and selects the desired number of sentences.
- The shortened text is returned.

6. Function: `get_Query(query)`
- This function is the entry point for retrieving Wikipedia content based on a query.
- It calls the `search_wikipedia()` function with the given query.
- If the result is not `None`, it returns the result.
- If there is an exception during the process, it returns `None`.
Overall, this code provides a way to search for a query on Wikipedia, retrieve the content of the Wikipedia page, clean the HTML, and shorten the text to a desired length. It utilizes the `requests` library for making API requests, `BeautifulSoup` for HTML parsing and cleaning, and regular expressions for text manipulation.

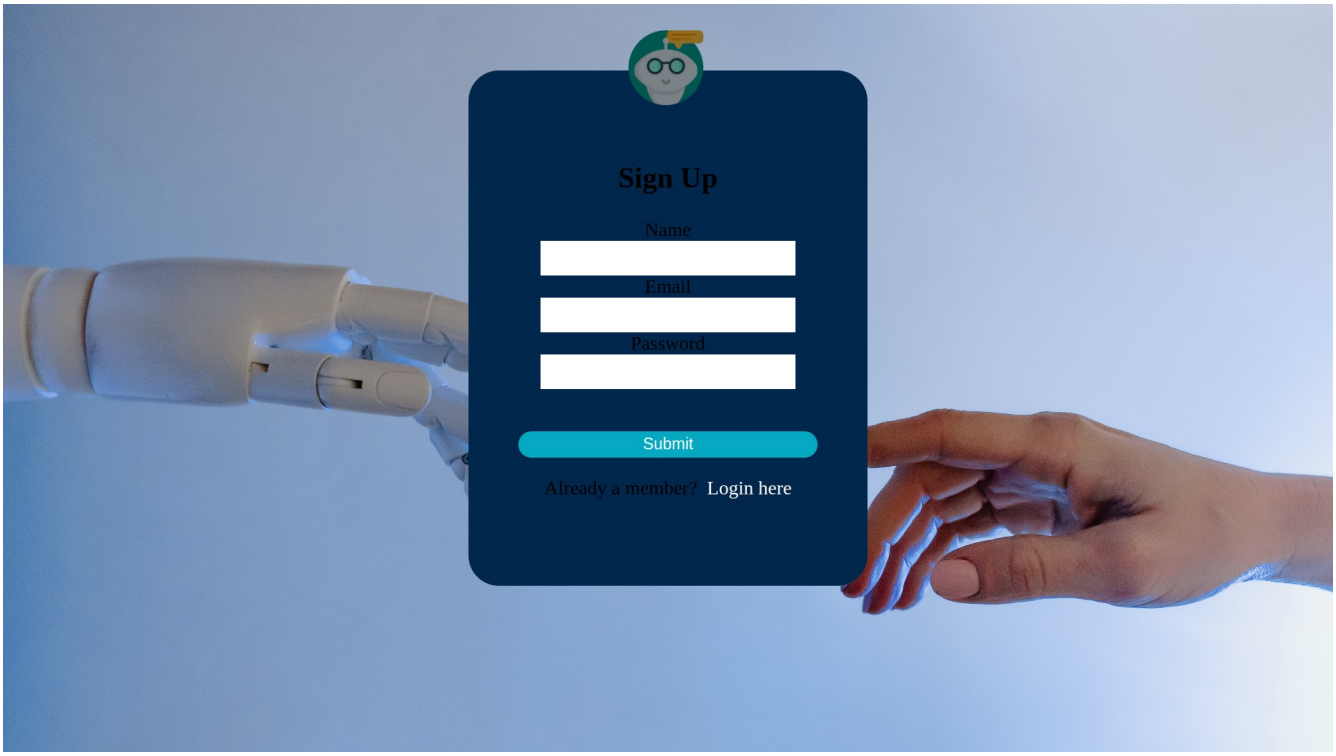
Steps to use :

-first run the **neo4jdatabase and then bot.py** *file and then follow the given flask app link*

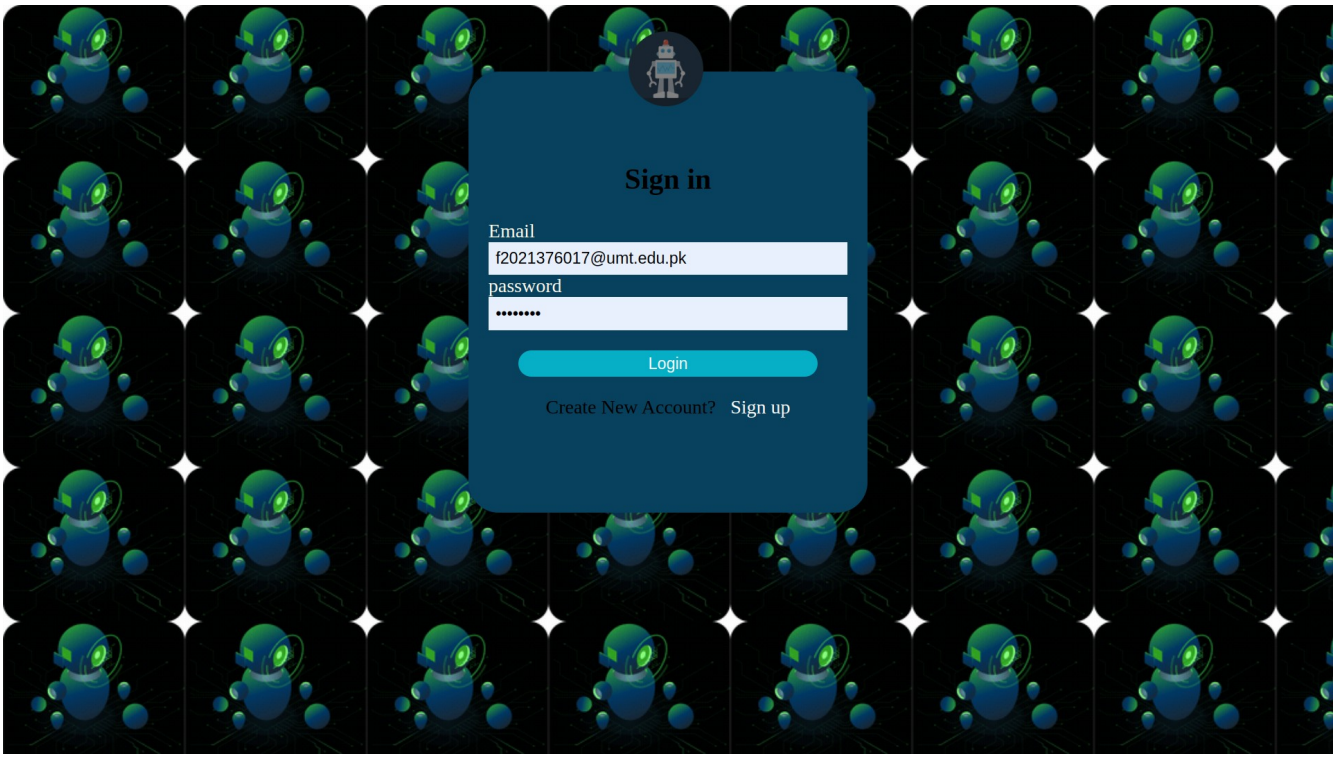
-Home page



-Sign up



-Sign in



-Chatting