CREATE A CHATBOT IN PYTHON

NAME: ARUN KRISHNA J
REG NO: 61772221T302

PHASE 2 Submission Document

TABLE OF CONTENTS:

Introduction:

Chatbots have become an essential component of modern communication, improving user experiences on websites, social networking platforms, and customer service systems. To construct an effective chatbot, we need to start with high-quality data and a thorough grasp of the dataset. This explains the critical

procedures for importing, cleaning, and analysing data as the foundation for the chatbot project.

Problem Statement:

Customers expect excellent service when using your app or website. They may lose interest in the app if they are unable to find an answer to a query they have. To avoid losing consumers and harming your bottom line, you must provide the best service possible while establishing a website or application.

Creating a Chatbot in Python: Data Preparation and Analysis

NLP(Natural Language Process):

NLP is a method for computers to intelligently analyse, comprehend, and derive meaning from human language. Developers can use NLP to organise and structure knowledge in order to execute tasks like automatic summarization, translation, named entity recognition, relationship

extraction, sentiment analysis, audio recognition, and topic segmentation.

Import necessary libraries

```
import io
import random
import string # to process standard python strings
import warnings
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import warnings
warnings.filterwarnings('ignore')
```

Reading in the corpus:

As an example, we'll use the Wikipedia page on chatbots as our corpus. Copy the contents of the page and save it as 'chatbot.txt' in a text file. You may, however, utilise any corpus of your choice.

```
f=open('chatbot.txt','r',errors = 'ignore')
raw=f.read()
raw = raw.lower()# converts to lowercase
```

The fundamental issue with text data is that it is all in string format. However, in order to execute the work, machine learning algorithms require some kind of numerical feature vector, so before we start with any NLP project we need to

pre-process it to make it ideal for working. Basic text pre-processing includes:

- Converting the entire text into **uppercase** or **lowercase**, so that the algorithm does not treat the same words in different cases as different
- **Tokenization**: Tokenization is just the process of transforming standard text strings into a list of tokens, or words that we truly desire. Sentence tokenizer can be used to find a list of sentences, whereas Word tokenizer can find a list of words in strings.

The NLTK data package includes a pre-trained Punkt tokenizer for English.

- Removing **Noise** i.e everything that isn't in a standard number or letter.
- Removing the **Stop words**. Sometimes, some extremely common words which would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely. These words are called stop words
- **Stemming**: The process of reducing inflected (or sometimes derived) words to their stem, base, or root form — typically a written word form — is known as stemming. For example, if we stemmed the phrases "Stems", "Stemming", "Stemmed", and "and Stemtization", the result would be a single word "stem".
- **Lemmatization**: A slight variant of stemming is lemmatization. The major difference between these is, that, stemming can often create non-existent words, whereas lemmas are actual words. So, your root stem,

meaning the word you end up with, is not something you can just look up in a dictionary, but you can look up a lemma. Examples of Lemmatization are that “run” is a base form for words like “running” or “ran” or that the word “better” and “good” are in the same lemma so they are considered the same.

Tokenisation

```
sent_tokens = nltk.sent_tokenize(raw)# converts to list of sentences
word_tokens = nltk.word_tokenize(raw)# converts to list of words
```

Preprocessing

We shall now define a function called LemTokens which will take as input the tokens and return normalized tokens.

```
lemmer = nltk.stem.WordNetLemmatizer()
#WordNet is a semantically-oriented dictionary of English included in NLTK.
def LemTokens(tokens):
    return [lemmer.lemmatize(token) for token in tokens]
remove_punct_dict = dict((ord(punct), None) for punct in string.punctuation)

def LemNormalize(text):
    return LemTokens(nltk.word_tokenize(text.lower().translate(remove_punct_dict)))
```

Keyword matching

Next, we shall define a function for a greeting by the bot i.e if a user's input is a greeting, the bot shall return a greeting response.ELIZA uses a simple keyword matching for greetings. We will utilize the same concept here.

```
GREETING_INPUTS = ("hello", "hi", "greetings", "sup", "what's up","hey",)
GREETING_RESPONSES = ["hi", "hey", "*nods*", "hi there", "hello", "I am glad! You are talking to me"]
def greeting(sentence):

    for word in sentence.split():
        if word.lower() in GREETING_INPUTS:
            return random.choice(GREETING_RESPONSES)
```

Generating Response

Bag of Words

After the initial preprocessing phase, we need to transform text into a meaningful vector (or array) of numbers. The bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

Why is it is called a "bag" of words? That is because any information about the order or structure of words in the document is discarded and the model is only **concerned with**

whether the known words occur in the document, not where they occur in the document.

The intuition behind the Bag of Words is that documents are similar if they have similar content. Also, we can learn something about the meaning of the document from its content alone.

For example, if our dictionary contains the words {Learning, is, the, not, great}, and we want to vectorize the text "Learning is great", we would have the following vector: (1, 1, 0, 0, 1).

TF-IDF Approach

A problem with the Bag of Words approach is that highly frequent words start to dominate in the document (e.g. larger score), but may not contain as much "informational content". Also, it will give more weight to longer documents than shorter documents.

One approach is to rescale the frequency of words by how often they appear in all documents so that the scores for frequent words like "the" that are also frequent across all documents are penalized. This approach to scoring is called Term Frequency-Inverse Document Frequency, or TF-IDF for short, where:

Term Frequency: is a scoring of the frequency of the word in the current document.

TF = (Number of times term t appears in a document)/(Number of terms in the document)

Inverse Document Frequency: is a scoring of how rare the word is across documents.

IDF = 1+log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

Cosine Similarity

Tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus

Cosine Similarity (d1, d2) = Dot product(d1, d2) / ||d1|| * ||d2||

where d1,d2 are two non zero vectors.

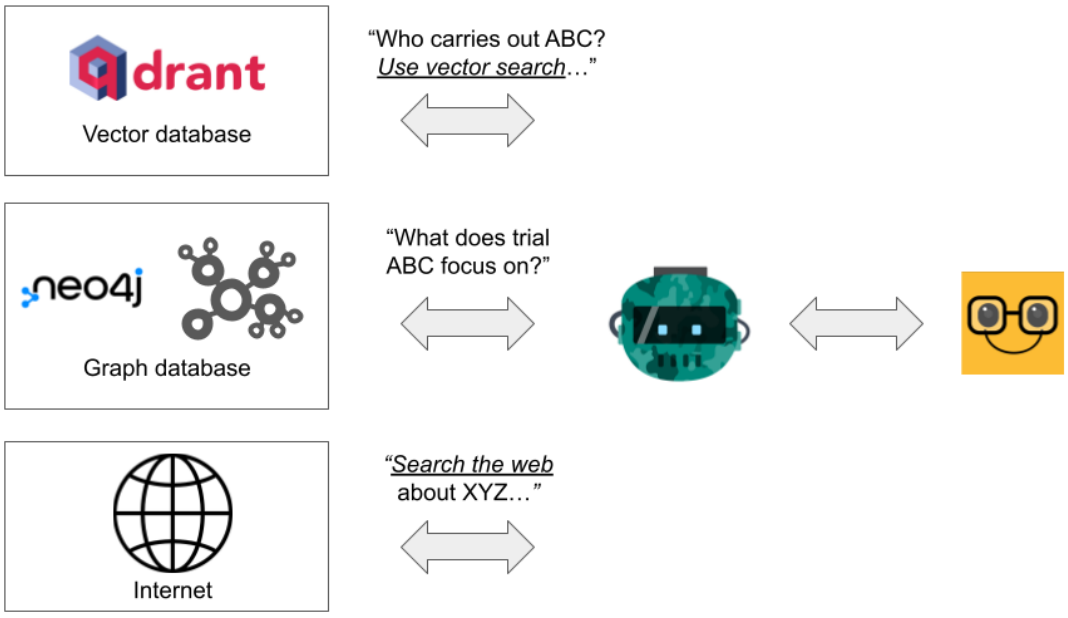
To generate a response from our bot for input questions, the concept of document similarity will be used. We define a function response which searches the user's utterance for one or more known keywords and returns one of several possible responses. If it doesn't find the input matching any of the keywords, it returns a response:" I am sorry! I don't understand you"

```
def response(user_response):
    robo_response=''
    sent_tokens.append(user_response)
    TfidfVec = TfidfVectorizer(tokenizer=LemNormalize,
stop_words='english')
    tfidf = TfidfVec.fit_transform(sent_tokens)
```

```
    vals = cosine_similarity(tfidf[-1], tfidf)
    idx=vals.argsort()[0][-2]
    flat = vals.flatten()
    flat.sort()
    req_tfidf = flat[-2]
    if(req_tfidf==0):
        robo_response=robo_response+"I am sorry! I don't
understand you"
        return robo_response
    else:
        robo_response = robo_response+sent_tokens[idx]
        return robo_response
```

Finally, we will feed the lines that we want our bot to say while starting and ending a conversation depending upon user's input.

Import the Dataset:



To build a chatbot, we first need a dataset.We can either collect conversational data or obtain a dataset from sources like Twitter, Reddit, or customer support interactions. Assuming that we have a CSV file with our dataset, we can import it using Python and Pandas:

```
import pandas as pd

#Load any dataset

df = pd.read_csv('your_dataset.csv')
```

Data Cleaning:

Data cleaning is essential to ensure your dataset is usable for training a chatbot. It involves tasks such as handling missing values, removing duplicates, and text preprocessing.

Handle Missing Values:

```
df.dropna(subset=['text_column'], inplace=True) # Remove rows with missing text data
```

Remove Duplicates:

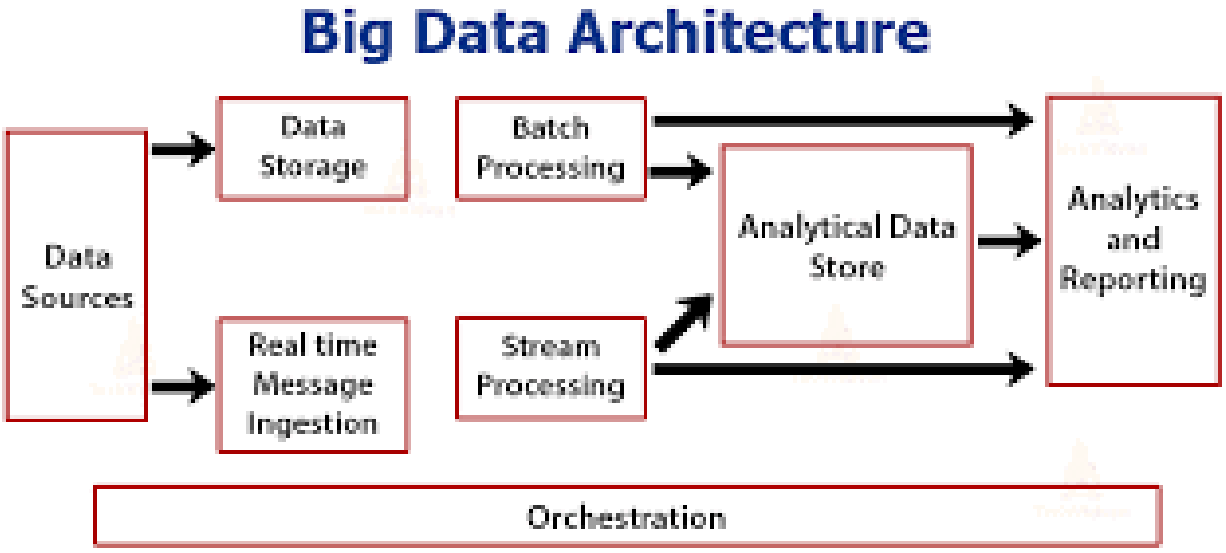
```
df.drop_duplicates(subset=['text_column'], inplace=True)
```

Text Preprocessing:

```
import re
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
# Define a function to preprocess text
def preprocess_text(text):
    # Remove special characters and digits
    text = re.sub(r'[^a-zA-Z]', ' ', text)
    # Convert to lowercase
    text = text.lower()
    # Tokenize and remove stopwords
    text = ' '.join([word for word in text.split() if word not in set(stopwords.words('english'))])
```

```
    # Stemming (if needed)
    stemmer = PorterStemmer()
    text = ' '.join([stemmer.stem(word) for word in text.split()])
    return text
df['cleaned_text'] = df['text_column'].apply(preprocess_text)
```

Data Analysis:



Understanding your dataset through data analysis is a crucial step in creating a chatbot. It provides insights and informs your chatbot's design.

Basic Analysis:

```
# Get basic statistics
num_samples = len(df)
```

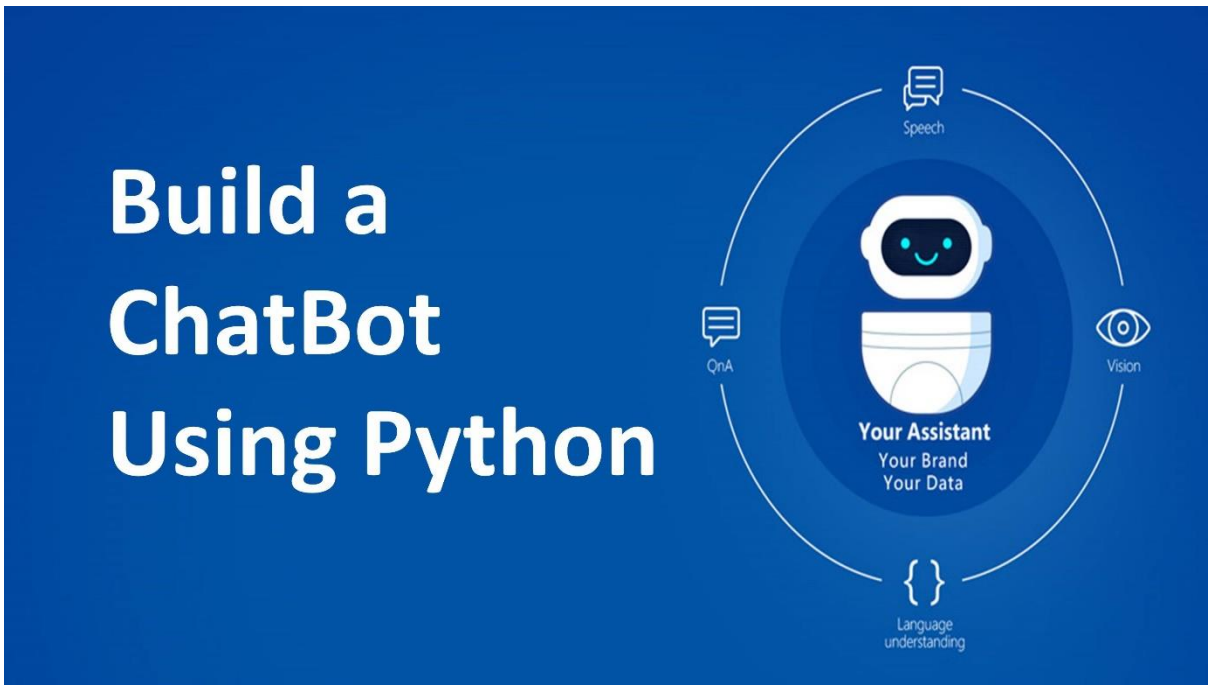
```
average_sentence_length = df['cleaned_text'].apply(lambda x: len(x.split())).mean()

# Print statistics

print(f"Number of samples: {num_samples}")

print(f"Average sentence length: {average_sentence_length}")
```

Visualization:

You can create visualizations to understand the distribution of data. For instance, you can visualize the sentence length distribution:

```
import matplotlib.pyplot as plt

# Visualize sentence length distribution

plt.hist(df['cleaned_text'].apply(lambda x: len(x.split())), bins=20)

plt.xlabel('Sentence Length')

plt.ylabel('Count')

plt.show()
```

Building the Chatbot:



With clean and analyzed data in hand, we can now proceed to build our chatbot.

This guide covers the initial steps of importing a dataset, cleaning the data, and performing data analysis. The next steps involve training our chatbot, which may depend on the specific chatbot technology we choose and its intended functionality.

Code and Dataset:

Code:

```
[*]: flag=True
def read_dataset(filename):
    dataset = {}
    with open(filename, 'r') as file:
        for line in file:
            question, answer = line.strip().split('?')
            dataset[question] = answer
    return dataset
def get_answer(question, dataset):
    return dataset.get(question, "I don't have the answer to that question.")

dataset = read_dataset("datafor.txt")

print("ROBO: My name is Robo. I will answer your queries about Chatbots. If you want to exit, type Bye!")
while(flag==True):
    user_response = input()
    user_response=user_response.lower()
    if(user_response!='bye'):
        if(user_response=='thanks' or user_response=='thank you' ):
            flag=False
            print("ROBO: You are welcome..")
        else:
            print("ROBO: "+get_answer(user_response,dataset))

    else:
        flag=False
        print("ROBO: Bye! take care..")
```

Output:

```
ROBO: My name is Robo. I will answer your queries about Chatbots. If you want to exit, type Bye!
 hi, how are you doing
ROBO:   i'm fine, how about yourself.
 i'm fine. how about yourself
ROBO:  i'm pretty good, thanks for asking.
 bye
ROBO: Bye! take care..
```

Dataset Link:

https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot

Conclusion:

Creating a chatbot is a multi-faceted endeavor that starts with data preparation and analysis. By importing a relevant dataset, cleaning the data, and analyzing it, we set a solid foundation for our chatbot project. These initial steps pave the way for a more efficient and effective chatbot, ultimately enhancing user experiences and facilitating automated interactions.

The subsequent steps of training, testing, and deploying your chatbot are equally important. Building a chatbot is a dynamic process that requires ongoing refinement and adaptation to meet our users' needs.