**INTERACTIVE CHATBOT APPLICATION FOR COMPUTER SCIENCE DOMAIN USING NLTK**

**ABSTRACT**

People interact with systems more and more through voice assistants and chat bots. The days of solely engaging with a service through a voice are over. These new modes of user interaction are aided in part by advancements in Artificial Intelligence and natural language processing.

This project aimed to implement a web-based chat bot to assist students to learn the computer science related subjects using tools that expose artificial intelligence methods such as natural language understanding. Allowing users to interact with the chat bot using natural language input and to train the chat bot using appropriate methods so it will be able to generate a response. The prototype was mainly implemented using NLTK library.

The produced prototype was found to be a very useful tool to justify the need of a modern method of interaction to learn computer science subjects. Students can ask their doubts in the chat bot and the results are returned instantly.

**DATA PRE-PROCESSING**

Text Pre-processing is a very important phase in any NLP based applications. Improper pre-processing of Text data will lead to inconsistencies in the result.

Different pre-processing techniques applied in our project are given below.

1. Case Folding (Lower Case)

Lowercasing ALL your text data, although commonly overlooked, is one of the simplest and most effective form of text pre-processing. It is applicable to most text mining and NLP problems and can help in cases where your dataset is not very large and significantly helps with consistency of expected output. Lower casing is very useful in search.

1. Sentence Tokenization

Sentences are retrieved from the different documents and stored in some variables. Sentence segmentation is done by sent\_tokenize() function in the NLTK.

1. Word Tokenization

Words are retrieved from the sentences and stored in the variable. It is done by word\_tokenize() function in the NLTK.

1. Lemmatization

Lemmatization on the surface is very similar to stemming, where the goal is to remove inflections and map a word to its root form. The only difference is that, lemmatization tries to do it the proper way. It doesn’t just chop things off, it actually transforms words to the actual root. It is done by WordNetLemmatizer.

1. Stop word Removal

Stop words are a set of commonly used words in a language. Examples of stop words in English are “a”, “the”, “is”, “are” and etc. The intuition behind using stop words is that, by removing low information words from text, we can focus on the important words instead.

**PROPOSED TECHNIQUE**

We have implemented the chat bot by using NLTK. **NLTK(Natural Language Toolkit)** is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries.

We have extracted text for different domain of computer science from Wikipedia and stored as separate document. We use text pre-processing techniques to store the words. These are ranked based on the TF-IDF ranking and the corresponding results are retrieved for query from user.

Detailed steps are described below

## Bag of Words

After the initial pre-processing phase, we need to transform the 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.

**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.

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

TF-IDF is a transformation applied to texts to get two real-valued vectors in vector space. We can then obtain the **Cosine**similarity of any pair of vectors by taking their dot product and dividing that by the product of their norms. That yields the cosine of the angle between the vectors. **Cosine similarity** is a measure of similarity between two non-zero vectors. Using this formula we can find out the similarity between any two documents d1 and d2.

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

where d1,d2 are two non zero vectors.

## Corpus

For our project, we will be using 9 files which we have extracted from the Wikipedia page as our corpus.

## Reading in the data

We will read the 9 text file and convert the entire file into a list of sentences and a list of words for further pre-processing.

## Generating Response

To generate a response from our bot for input questions, the concept of document similarity will be used.

This is done by TFIDF vectorizer to convert a collection of raw documents to a matrix of TF-IDF features, and cosine similarity which is used to find the similarity between words entered by the user and the words in the corpus.

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”