“Develop a MLAI based summarization and data aggregation for generating insights from “Energy Data feed

“(Schlumberger's New Year Hackathon Shaastra 2023 Indian Institute of Technology (IIT), Madras)”

This code uses a combination of web scraping, natural language processing (NLP), and text summarization techniques to extract and summarize the energy-related content from the website **https://climate.mit.edu/explainers/carbon-capture**.

1. The code starts by importing the necessary libraries: requests, BeautifulSoup, NLTK, NumPy and NetworkX.
2. It then uses the requests library to send a GET request to the website, and BeautifulSoup to parse the HTML content of the website.
3. After getting the text content from the website, it uses NLTK's sent_tokenize() function to tokenize the text into sentences.
4. The code then use NLTK's word_tokenize() and pos_tag() functions to tokenize the text into words and assigns POS tags to each word.
5. Then it filters out the words which are nouns and noun phrases by checking the POS tag, an d assigns them as scientific_terms.
6. Next, it creates a list to hold the sentences that contain the scientific_terms, and iterates over the sentences and check if they contain any of the scientific_terms, if they do it append it to the energy_sentences list.
7. Then it creates a list of stopwords using NLTK's stopwords.words() function and defines a function to calculate the similarity between sentences. This function takes two sentences as input and calculates the cosine similarity between them after removing the stopwords.
8. The code then creates a similarity matrix

This can be termed as the synoptic explanation of the code.

The step-by-step explanation of code includes:

```
import requests
from bs4 import BeautifulSoup
import nltk
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
from nltk.cluster.util import cosine_distance

```

The code imports the necessary libraries to scrape a website, perform natural language processing (NLP) and text summarization.

1. The requests library is used to send HTTP requests to a website and retrieve its content.
2. The BeautifulSoup library is used to parse the HTML content of a website and extract the text.
3. The NLTK (Natural Language Toolkit) library is used for various NLP tasks such as tokenizing text, POS tagging and removing stopwords.
4. The sent_tokenize function from NLTK is used to tokenize the text into sentences.
5. The stopwords corpus from NLTK is used to remove stopwords from the text.
6. The cosine_distance function from NLTK's cluster.util package is used to calculate the similarity between sentences based on their word vectors.

Together, these libraries and functions are used to scrape a website, extract its text content, tokenize it into sentences, and calculate the similarity between these sentences to generate a summary.

```
# Crawl the website and extract the content
url = "https://climate.mit.edu/explainers/carbon-capture"
page = requests.get(url)
soup = BeautifulSoup(page.content, "html.parser")
text = soup.get_text()

```

It then uses the BeautifulSoup library to parse the HTML content of the website, and extract the text from the website.

The line "soup = BeautifulSoup(page.content, "html.parser")" uses the html.parser to parse the HTML content of the website, and "text = soup.get_text()" is used to extract all the text from the website.

This code is used to retrieve the content of the website and extract the text so that it can be processed further for NLP tasks.

```

# Tokenize the text into sentences
sentences = sent_tokenize(text)

# Define the scientific_terms to look for
tokens = nltk.word_tokenize(text)
# POS tagging
tagged_tokens = nltk.pos_tag(tokens)
# extract scientific terms
scientific_terms = [word for word, pos in tagged_tokens if pos in ["NN","NNS","NNP","NNPS"]]


```

This set code uses NLTK to perform natural language processing (NLP) on the text that was extracted from the website.

1. The first line uses the sent_tokenize() function from NLTK to tokenize the text into sentences. This function takes the text as input and returns a list of sentences.

2. The next step is to tokenize the text into words and assign POS tags to each word. The word_tokenize() function from NLTK is used to tokenize the text into words and the pos_tag() function is used to assign POS tags to each word.
3. Then it filters out the words which are nouns and noun phrases by checking the POS tag, and assigns them as scientific_terms. A list comprehension is used to filter out the words which are nouns and noun phrases and assign them to the scientific_terms list.

This code is used to tokenize the text into sentences and extract scientific terms from the text based on POS tagging. These sentences and scientific terms will be used later in the code to generate a summary.

```
# Create a list to hold the sentences that contain the scientific_terms
energy_sentences = []

# Iterate over the sentences and check if they contain any of the scientific_terms
for sentence in sentences:
    for keyword in scientific_terms:
        if keyword in sentence:
            energy_sentences.append(sentence)
            break

# Create a list of stopwords
stop_words = stopwords.words("english")
```

This code is used to filter out the sentences that contain the scientific terms, which will be further used for text summarization.

1. It creates an empty list named energy_sentences, which will be used to hold the sentences that contain the scientific terms.
2. It then iterates over each sentence in the sentences list, and for each sentence, it iterates over each keyword in the scientific_terms list.
3. For each keyword, it checks if the keyword is present in the current sentence. If it is, the current sentence is appended to the energy_sentences list and the inner loop is broken to avoid duplicates.
4. Then, it creates a list of stopwords in the English language using the stopwords.words() function from NLTK. These stopwords will be used later in the code to remove them from the sentences for calculating the similarity between sentences.

```
# Create a function to calculate the similarity between sentences
def sentence_similarity(sent1, sent2, stopwords=None):
    if stopwords is None:
        stopwords = []
    sent1 = [w.lower() for w in sent1]
    sent2 = [w.lower() for w in sent2]
    all_words = list(set(sent1 + sent2))
    vector1 = [0] * len(all_words)
    vector2 = [0] * len(all_words)
    for w in sent1:
        if w in stopwords:
            continue
        vector1[all_words.index(w)] += 1
    for w in sent2:
        if w in stopwords:
            continue
        vector2[all_words.index(w)] += 1
    return 1 - cosine_distance(vector1, vector2)
```

function named sentence_similarity() that is used to calculate the similarity between two sentences.

1. The function takes in two sentences, sent1 and sent2, and a list of stopwords as input. If the stopwords input is not provided, it defaults to an empty list.
2. It converts all the words in both sentences to lowercase using list comprehension.
3. It then creates a list of all unique words present in both sentences using set().
4. It creates two vectors, vector1 and vector2, each with the same length as the unique words list.
5. It then iterates over the words in the first sentence, and for each word, it checks if it is a stopword. If it is, it continues to the next word. If not, it increments the count of the word in vector1 at the index of the word in the unique words list.
6. It then iterates over the words in the second sentence, and for each word, it checks if it is a stopword. If it is, it continues to the next word. If not, it increments the count of the word in vector2 at the index of the word in the unique words list.
7. It then calculates the cosine similarity between vector1 and vector2 using the cosine_distance() function from NLTK's cluster.util package and returns the similarity score.

This function is used to calculate the similarity between two sentences based on the presence or absence of words in them after removing the stop words.

```
# Create a similarity matrix
similarity_matrix = [[sentence_similarity(sent1, sent2, stop_words) for sent1 in energy_sentences] for sent2 in energy_sentences]

import numpy as np

# Convert the similarity matrix list to a numpy array
similarity_matrix = np.array(similarity_matrix)
```

1. It uses a nested list comprehension to iterate over every sentence in the energy_sentences list. For each sentence, it calculates its similarity with every other sentence in the energy_sentences list using the sentence_similarity() function, and assigns the similarity score to the corresponding position in the similarity matrix.
2. The similarity matrix is a 2-dimensional array where each element (i, j) represents the similarity score between sentence i and sentence j.
3. Then it imports numpy library and converts the similarity matrix from a list to a numpy array, this is done because numpy provides more efficient mathematical operations than native python.

This similarity matrix will be used later in the code to generate a summary of the text by identifying the most important sentences.

```
65
66 # Use the similarity matrix to generate a summary
67 import networkx as nx
68 nx_graph = nx.from_numpy_array(similarity_matrix)
69 scores = nx.pagerank(nx_graph)
70 ranked_sentences = sorted(((scores[i], s) for i, s in enumerate(energy_sentences)), reverse=True)
71
72 # Select the top N sentences with the highest scores
73 N =10    #default value is set to summarize the content with n number also can be inputed
74 top_sentences = [ranked_sentences[i][1] for i in range(N)]
75
76 #
77 # Join the sentences together to form the summary
78 summary = " ".join(top_sentences)
79 print(summary,end='')
80 print()
81 print("from :",url)
```

1. It imports the networkx library and creates a graph object from the similarity matrix using the from_numpy_array() function.
2. It then uses the pagerank() function from networkx to calculate the importance of each sentence in the graph. The pagerank algorithm assigns a score to each sentence based on the similarity of the sentence with other sentences, and the importance of the sentences it links to.
3. It then sorts the sentences based on their scores in descending order and stores the result in the ranked_sentences list.
4. It then selects the top N sentences with the highest scores, where N is the number of sentences that you want to include in the summary. The default value is 10 but it can also be inputed.
5. It then joins the top sentences together to form the summary using the join() method.
6. Finally, it prints the summary and the website from which the content is scraped.

summary of the text by selecting the most important sentences based on the similarity matrix and page rank algorithm.
