Individual Information Retrieval Task for ICS 2205 Project

Document indexing

In [ ]:
```
# Installing nltk library
# pip install nltk
```

1. Parse the document to extract the data in the XML’s < raw > tag

In [ ]:
```
import nltk
import string
from xml.etree.ElementTree import parse
import os
import re

# Getting a list of all files in the folder
folder = "docs"
files = os.listdir(folder)

# Looping over the files in the folder
for file in files:
    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw = raw_tag.text

    # Remove all punctuation from the raw text using a regular expression
    raw_text = re.sub(r'[{}]'.format(string.punctuation), '', raw)

    # Printing the extracted text without punctuation
    print(raw_text)
```

2. Tokenise the documents’ content

In [ ]:
```
# Imports
from nltk.tokenize import word_tokenize

for file in files:

    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw = raw_tag.text

     # Remove all punctuation from the raw text using a regular expression
    raw_text = re.sub(r'[{}]'.format(string.punctuation), '', raw)

    # Tokenizing the text into words or tokens
    tokens = word_tokenize(raw_text)

    # Printing the tokens
    print(tokens)
```

3. Perform case-folding, stop-word removal and stemming

Case Folding:

- **Lower Case:**

In [ ]:
```
for file in files:

    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw = raw_tag.text

    # Remove all punctuation from the raw text using a regular expression
    raw_text = re.sub(r'[{}]'.format(string.punctuation), '', raw)

    # Using the .lower() function to convert the string to lowercase
    lowercase = raw_text.lower()
    # Printing lowercase text
    print(lowercase)
```

- **Upper Case:**

In [ ]:
```
for file in files:

    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw = raw_tag.text

     # Remove all punctuation from the raw text using a regular expression
    raw_text = re.sub(r'[{}]'.format(string.punctuation), '', raw)

    # Using the .upper() function to convert the string to uppercase
    uppercase = raw_text.upper()
    print(uppercase)
```

Stop-word removal:

- In this example, the nltk.corpus.stopwords.words() function is used to get a list of English stop words. The stop words are removed from the list of tokens using a list comprehension.

In [ ]:
```
for file in files:

    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw = raw_tag.text

    # Remove all punctuation from the raw text using a regular expression
    raw_text = re.sub(r'[{}]'.format(string.punctuation), '', raw)

    # Getting a list of English stop words
    stop_words = nltk.corpus.stopwords.words("english")

    # Splitting the text into tokens
    tokens = nltk.tokenize.word_tokenize(raw_text)

    # Removing the stop words from the tokens
    filtered_tokens = [token for token in tokens if token not in stop_words]

    # Printing the resulting list of tokens
    print(filtered_tokens)
```

Stemming:

- In this example, the Porter stemming algorithm was used

> The Porter stemming algorithm (or 'Porter stemmer') is a process for removing the commoner morphological and inflexional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems. [Martin, P. (1999). The Porter Stemming Algorithm. [Online]. Available: https://tartarus.org/martin/PorterStemmer]

In [ ]:
```
# Imports
from nltk.stem import PorterStemmer

for file in files:

    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw = raw_tag.text

    # Remove all punctuation from the raw text using a regular expression
    raw_text = re.sub(r'[{}]'.format(string.punctuation), '', raw)

    # Creating a stemmer object
    stemmer = PorterStemmer()

    # Stemming the raw text
    stemmed_text = [stemmer.stem(word) for word in raw_text.split()]

    # Printing stemmed raw text
    print(stemmed_text)
```

4. Build the term by document matrix containing the T F.IDF weight for each term within each document

In [ ]:
```
import math
import numpy as np
from nltk.tokenize import word_tokenize

def compute_tfidf(documents):
    # Tokenize the documents
    tokens = [word_tokenize(doc) for doc in documents]

    # Compute the term frequency (TF) for each term
    tf = []
    for doc in tokens:
        term_freq = {}
        for term in doc:
            if term in term_freq:
                term_freq[term] += 1
            else:
                term_freq[term] = 1
        tf.append(term_freq)

    # Compute the inverse document frequency (IDF) for each term
    idf = {}
    num_docs = len(documents)
    for doc in tokens:
        for term in doc:
            if term in idf:
                continue
            df = sum(1 for d in tokens if term in d)
            idf[term] = math.log(num_docs / df)

    # Compute the TF-IDF weight for each term
    tfidf = []
    for doc in tf:
        doc_tfidf = {}
        for term, freq in doc.items():
            doc_tfidf[term] = freq * idf[term]
        tfidf.append(doc_tfidf)

    # Create the term-by-document matrix
    terms = sorted(idf.keys())
    term_by_doc_matrix = np.zeros((len(terms), num_docs))
    for i, term in enumerate(terms):
        for j, doc in enumerate(tfidf):
            if term in doc:
                term_by_doc_matrix[i, j] = doc[term]

    return terms, term_by_doc_matrix

# List of documents
documents = []

for file in files:

    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw_text = raw_tag.text

    # Adding the document to the list of documents
    documents.append(raw_text)

# Compute the term-by-document matrix
terms, term_by_doc_matrix = compute_tfidf(documents)

# Print the TF-IDF weights for each term in the matrix
for i, row in enumerate(term_by_doc_matrix):
    term = terms[i]
    print(f'TF-IDF weights for term "{term}": {row}')
    print('\n')
```

Querying Component

- ### Get a user query – note that it can be set within the notebook directly. Into a variable named query;

Outputting all the quries:

In [ ]:
```
import xml.etree.ElementTree as ET

# Get the current working directory
cwd = os.getcwd()
# Construct the path to the "queries" folder
folder = os.path.join(cwd, "queries")
# Get a list of all files in the "queries" folder
files = os.listdir(folder)

# Loop over the files in the "queries" folder
for file in files:
    # Parse the .naf file using ElementTree
    file_path = os.path.join(folder, file)
    tree = ET.parse(file_path)
    # Find the <raw> tag
    raw_tag = tree.find("raw")
    # Extract the contents of the <raw> tag
    query_raw_text = raw_tag.text
    # Print the extracted text
    print(query_raw_text)
```

Selecting one of the quries (choose from above):

In [ ]:
```
# Get the user's query
query = input("Enter your query: ")

# Print the user's query
print("Your query:", query)
```

- ## Preprocess the user query (tokenisation, case-folding, stop-word removal and stemming)

In [ ]:
```
# Imports
import re

# Converting the query to lowercase
query = query.lower()

# Tokenizing the query into words or tokens
tokens = word_tokenize(query)

# Getting a list of English stop words
stop_words = nltk.corpus.stopwords.words("english")
# Removing the stop words from the tokens
filtered_tokens = [token for token in tokens if token not in stop_words]

# Creating a stemmer object
stemmer = PorterStemmer()
# Stemming the tokens
query_tokens = [stemmer.stem(token) for token in tokens]

# Print the preprocessed tokens
print("Preprocessed query:", query_tokens)
```

- ## Use cosine similarity to calculate the similarity between the query and each document

Calculating cosine similarity between query and documents

In [ ]:
```
from collections import Counter

# Get the current working directory
cwd = os.getcwd()
# Construct the path to the "docs" folder
folder = os.path.join(cwd, "docs")
# Get a list of all files in the "docs" folder
files = os.listdir(folder)

def cosine_similarity(query_tokens, doc_tokens):
    # Create a Counter object for each list of tokens
    query_counter = Counter(query_tokens)
    doc_counter = Counter(doc_tokens)

    # Create a list of unique words present in either list of tokens
    unique_words = set(query_tokens).union(set(doc_tokens))

    # Calculate the dot product of the frequency vectors
    dot_product = sum(query_counter[word] * doc_counter[word] for word in unique_words)

    # Calculate the Euclidean length of the frequency vectors
    query_length = sum(query_counter[word]**2 for word in unique_words)
    doc_length = sum(doc_counter[word]**2 for word in unique_words)

    # If either length is 0, return 0 as the cosine similarity
    if query_length == 0 or doc_length == 0:
        return 0

    # Calculate the cosine similarity as the dot product of the frequency vectors divided by the product of their Euclidean lengths
    cosine_sim = dot_product / (math.sqrt(query_length) * math.sqrt(doc_length))

    return cosine_sim

# Iterate over all the documents in the folder
for file in files:
    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw_text_docs = raw_tag.text

    # Converting the raw_text_docs to lowercase
    raw_text_docs = raw_text_docs.lower()

    # Tokenizing the document into words or tokens
    tokens_docs = word_tokenize(raw_text_docs)

    # Removing stop words from the tokens
    filtered_tokens_docs = [token for token in tokens_docs if token not in stop_words]

    # Stemming the tokens
    stemmed_tokens_docs = [stemmer.stem(token) for token in filtered_tokens_docs]

    # Remove duplicates from the list
    no_duplicates = []
    for token in stemmed_tokens_docs:
        if token not in no_duplicates:
            no_duplicates.append(token)

    # Calculate the cosine similarity between the query and the current document
    cosine_sim = cosine_similarity(query_tokens, no_duplicates)
    # Print the cosine similarity between the query and the current document
    print("File: " f"{file} - {cosine_sim}\n")
```

- ## Output the list of documents as a ranked list

In [ ]:
```
results = []

for file in files:
    # Parsing the .naf files using nltk's ElementTree parser
    file_path = os.path.join(folder, file)
    tree = parse(file_path)

    # Finding the <raw> tag
    raw_tag = tree.find("raw")
    # Extracting contents from <raw> tag
    raw_text_docs = raw_tag.text

    # Converting the raw_text_docs to lowercase
    raw_text_docs = raw_text_docs.lower()

    # Tokenizing the document into words or tokens
    tokens_docs = word_tokenize(raw_text_docs)

    # Removing stop words from the tokens
    filtered_tokens_docs = [token for token in tokens_docs if token not in stop_words]

    # Stemming the tokens
    stemmed_tokens_docs = [stemmer.stem(token) for token in filtered_tokens_docs]

    # Remove duplicates from the list
    no_duplicates = []
    for token in stemmed_tokens_docs:
        if token not in no_duplicates:
            no_duplicates.append(token)

    # Calculate the cosine similarity between the query and the current document
    cosine_sim = cosine_similarity(query_tokens, no_duplicates)

    # Add the file name and cosine similarity to the results list
    results.append((file, cosine_sim))

# Sort the results list by cosine similarity
ranked_results = sorted(results, key=lambda x: x[1], reverse=True)

# Iterate over the ranked results and print the file name and cosine similarity
for result in ranked_results:
    file, cosine_sim = result
    print(f"File: {file} - Similarity: {cosine_sim}\n")
```