```
import nltk
import os
import string
import numpy as np
import math
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from collections import Counter
from pathlib import Path
```

```
!pip install num2words
```

```
Requirement already satisfied: num2words in /usr/local/lib/python3.6/dist-packages (0
Requirement already satisfied: docopt>=0.6.2 in /usr/local/lib/python3.6/dist-package
```

```
import num2words
from num2words import num2words
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

```
path = Path('/content/drive/My Drive/assignment_1_mod')
```

```
cd /content/drive/My Drive/assignment_1_mod
```

```
/content/drive/My Drive/assignment_1_mod
```

```
dataset = os.listdir(path)
print(dataset)
```

```
['d1.txt', 'd3.txt', 'd2.txt', 'd4.txt', 'd5.txt']
```

```
def print_document(id):
    print(dataset[id])
    file = open(dataset[id], 'r')
    text = file.read().strip()
    file.close()
    print(text)
```

Preprocessing of dataset

```
def lowercase(inputs):
    return np.char.lower(inputs)
```

```
nltk.download('stopwords')
```

```
from nltk.corpus import stopwords
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
def stopwords_remover(inputs):
    stop_words = stopwords.words('english')
    words = word_tokenize(str(inputs))
    modified_text = ""
    for word in words:
        if word not in stop_words and len(word) > 1:
            modified_text = modified_text + " " + word
    return modified_text


def punctuation_remover(inputs):
    symbols = "!\"#$%&()*+-./:;<=>?@[\]^_`{|}~\n"
    for i in range(len(symbols)):
        inputs = np.char.replace(inputs, symbols[i], ' ')
        inputs = np.char.replace(inputs, "  ", " ")
    inputs = np.char.replace(inputs, ',', '')
    inputs = np.char.replace(inputs, "'", "")
    return inputs


def stemming(inputs):
    stemmer = PorterStemmer()
    tokens = word_tokenize(str(inputs))
    modified_text = ""
    for word in tokens:
        modified_text = modified_text + " " + stemmer.stem(word)
    return modified_text


def convert_numbers(inputs):
    tokens = word_tokenize(str(inputs))
    modified_text = ""
    for word in tokens:
        try:
            word = num2words(int(word))
        except:
            a = 0
        modified_text = modified_text + " " + word
    modified_text = np.char.replace(modified_text, "-", " ")
    return modified_text


def preprocess(text):
    text = lowercase(text)
    text = punctuation_remover(text)
    text = stopwords_remover(text)
    text = convert_numbers(text)
    text = stemming(text)
    text = punctuation_remover(text)
    text = convert_numbers(text)
    text = stemming(text)
    text = punctuation_remover(text)
```

```
    text = stopwords_remover(text)
    return text


nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
True
```

```
processed_text = []
for i in dataset:
    file = open(i, 'r', errors='ignore')
    text = file.read()
    file.close()
    processed_text.append(word_tokenize(str(preprocess(text))))


processed_text


N = len(dataset)
# N = 5
```

'processed_text' contains a list of lists of the words occurring in each document

Creating document frequency(nj in idf)

```
nj = {}
# nj is a dictionary that is of the format key : {}
# key is the word from a list and {} is a set containing all the documents in which 'key'
for i in range(N):
    tokens = processed_text[i]
    for word in tokens:
        try:
          # when the set contains some element in it add a new element to the set
            nj[word].add(i)
        except:
          # when the set is empty
            nj[word] = {i}


# counting the number of elements present in the set part if nj dictionary.
# This gives the number of documents in which a word is occurring.
for i in nj:
    nj[i] = len(nj[i])
```

Now nj contains a dictionary representing a word and the number of documents in which that wor in nj dictionary should give the entire set of words that are used in our dataset.

```
vocab_size = len(nj)


# creating a list of vocab words of our dataset
```

```
total_vocab = [x for x in nj]

# This function accesses the nj dictionary and gives the document count for the word passe
def document_frequency(word):
    counts = 0
    try:
        counts = nj[word]
    except:
        pass
    return counts
```

Calculating tf-idf

```
doc = 0
tf_idf = {}
# tf_idf is a dictionary that is of the form (document number, token) : tf-idf value
for i in range(N):
    tokens = processed_text[i]
    # A Counter is a container that keeps track of how many times equivalent values are ad
    # with key value as token and value as counts
    counter = Counter(tokens)
    words_count = len(tokens)

    for token in np.unique(tokens):
        tf = counter[token]/words_count
        df = document_frequency(token)
        idf = np.log((N+1)/(df+1))

        tf_idf[doc, token] = tf*idf

    doc += 1

tf_idf
```

Finding similarity

```
# function to calculate the cosine similarity
def cosine_sim(x,y):
    similarity = np.dot(x, y)/(np.linalg.norm(x)*np.linalg.norm(y))
    return similarity


#  'tf_idf_matrix' is a matrix initialized to zeros. The matrix tf_idf_matrix gives a tabu
#  arranged according to the documents to which they belong to.
# Each row of tf_idf_matrix represents a document and each column of tf_idf_matrix represe

tf_idf_matrix = np.zeros((N, vocab_size))
for i in tf_idf:
    try:
        # extract the token from the key value of tf_idf dictionary and put it into the va
        cell = total_vocab.index(i[1])
```

```
            cell = total_vocab.index(i[1])
            # Inserting the calculated tf_idf value of the token into a position in the D matr
            tf_idf_matrix[i[0]][cell] = tf_idf[i]
        except:
            pass


    tf_idf_matrix


    # Function for performing tf-idf calculations on the query document. Same as that of tf_id
    def query_document_vector(tokens):
        query_tf_idf_matrix = np.zeros((len(total_vocab)))
        counter = Counter(tokens)
        words_count = len(tokens)

        for token in np.unique(tokens):
            tf = counter[token]/words_count
            df = document_frequency(token)
            idf = math.log((N+1)/(df+1))

            try:
                cell = total_vocab.index(token)
                query_tf_idf_matrix[cell] = tf*idf
            except:
                pass
        return query_tf_idf_matrix


    # Function that generates a list of cosine similarity values for the query
    def cosine_similarity(query):
        preprocessed_query_document = preprocess(query)
        tokens = word_tokenize(str(preprocessed_query_document))

        print("\nGiven Query:", query)
        print("")

        # A list for storing the cosine similarities generated between documents
        cosine_values = []
        # creating tf_idf vector for query document
        query_td_idf_vector = query_document_vector(tokens)

        for xi in tf_idf_matrix:
          # Calculating cosine similarities between (d1, d_query) to (d4, d_query)
            cosine_values.append(cosine_sim(query_td_idf_vector, xi))

        # Sorting the 'cosine_values' list and returning it in the form of corresponding sorte
        output = np.array(cosine_values).argsort()[::-1]
        # 'output' is a list that contains the document numbers sorted(Descending) according t
        #
        return output
```

Instead of using the query document as a txt file I've used the contents of d_query.txt in my 'cosin

```
    sorted_similarites = cosine_similarity("java coffee mocha")
```

```
Given Query: java coffee mocha
```

```
# Printing the document with maximum cosine similarity
print_document(sorted_similarites[0])
```

```
d4.txt
Java coffee refers to coffee beans produced in the Indonesian island of Java.  The In

The coffee is primarily grown on large estates that were built by the Dutch in the 18

These estates transport ripe cherries quickly to their mills after harvest. The pulp

This coffee is prized as one component in the traditional "Mocca Java" blend, which p
```