Readme File

Initially, follow the below sequence for executing the code

- Upload all the dataset files on google colab.
- Change the directory to the drive folder where all the files are present as displayed below:

```
os.chdir('/content/drive/MyDrive/Assignment-2-IR/')
```

Question 1:

This question is in the code file **'IR_A2_Q1.ipynb'**

Pre-processing:

- Import libraries then Read the dataset
- We first tokenize the terms and then take each term of the sentence and convert it into lower case

```
def lower_case(text):
    lower_case_text = text.lower()
    return lower_case_text

def tokenize(text):
  tokenizer = TweetTokenizer()
  token_list = tokenizer.tokenize(text)
  return token_list
```

- Then we remove the special characters and extra blank space
- After that, we remove the stop words

```
def stop_word(text):
  sentence = []
  stop_words = set(stopwords.words("english"))
  for w in text:
        if w not in stop_words:
            sentence.append(w)
        else:
          continue
  return sentence

def remove_punc(text):
  punc_tokenizer = nltk.RegexpTokenizer(r"\w+")
  text = punc_tokenizer.tokenize(text)
  return text

def tokenizeReg(text):
    tok=RegexpTokenizer('[A-Za-z0-9]?\w+')
    return tok.tokenize(text)
```

- After that, we lemmatize the term.

```
[12] def lemmatization(file):
       lemmatizer = WordNetLemmatizer()
       for i in range(0,len(file)):
         lemma = lemmatizer.lemmatize(file[i])
         file[i] = lemma
       return file
```

For each part of this question that needs to be implemented, the question part is mentioned using a text cell at the beginning of the question part in the notebook file. These subparts can be executed after performing the pre-processing steps.

Assumptions:

- We're not using any stemming here instead lemmatization is used

Methodology:

- Firstly, the pre-processing is done on all the files and an we created a jaccard coefficient function which is taking query as an input, the query is preprocessed using the same that we have done for the input part the jaccard coefficient is find using the len of intersection between the quey and the content set by the len of the intersection between the union and query set
- For TF- IDF part, first we created the inverted index which is giving two things one is the words that arec present in the files and the other is the global content which is a dictionary that store the content of each document corresponding to its document Id. After that we found the vocab and the number of documents which are present in the data set and after that in a finalDict we store the word count of each document and then we found out the document frequency and after that we implement the 5 types of the tf-idf matrix mentioned in the assignment at the end we are finding the score using a scoring function and the which is taking model and numbe of documents as the input and after finding out that we are getting the top 5 relevant documents.

Question 2:

This question is in the code file **‘Assignment_2_Q2.ipynb’**
Follow the cell sequence for the execution.

Pre-processing:

- Import libraries
- Read the dataset

So these are the preprocessing, which we are doing in this question.

Methodology:

Created file for achieving the maximum DCG value which would when the order of document would be in the decreasing value of the relevance of the DCG

Code snippets for above part:

```
def keyRelBasedList(df):
  temp_list = []
  for j in range(len(df)):
    temp_list.append((list(df.index)[j],list(df[0])[j]))

  return temp_list

idxRelList = keyRelBasedList(df_txt_q4)

def sortList(tmp_list):
  return tmp_list.sort(key= lambda t: tmp_list[1],reverse=True)

sortList(idxRelList)
```

Then just used the above data for calculating maximum DCG value:

```
def maxDcg(tmp_list):
  max_dcg = 0;
  for j in range(len(tmp_list)):
    if j == 0:
            max_dcg = tmp_list[j][1]
    else:
            max_dcg += tmp_list[j][1]/math.log2(j+1)
  return max_dcg
```

Then just nDcg formula used for it's calculation and documents are limited based on the parameter stopping index as shown below

```
def nDCG(df,stoppingIdx):
    tmp_list = []
    idx_list =  list(df.index)
    rel_list = list(df[0])
    for j in range(len(idx_list)):
        tmp_list.append((idx_list[j],rel_list[j]))

    dcg = 0
    for j in range(len(tmp_list)):
      if j == 0:
              dcg = tmp_list[j][1]
      else:
              dcg += tmp_list[j][1]/math.log2(j+1)
      if( j == stoppingIdx ):
          break
    dcg = dcg/max_dcg

    return dcg
```

Assumptions: No Assumptions

Question 3:

This question is in the code file **'Assignment2_Q3.ipynb'**
Follow the cell sequence for the execution.

Pre-processing:

- Import libraries
- Read the dataset
- Remove stop words like is, are
- Replacing special symbols with space
- Removing extra spaces
- Performing lemmatization
- converting whole text to lower case
- word tokenization

So these are the preprocessing, which we are doing in this question.

Methodology:

1. Unzipped the given dataset
2. Read the file of these classes `['talk.politics.misc','comp.graphics', 'sci.space','rec.sport.hockey', 'sci.med' ]`
3. Performed pre-processing on all the files read
4. Created function for calculating the class words, word count in class and tF-Icf for the term is calculated.

```
def createClassBasedWords(data,class_list):
  temp = {}
  for j in range(len(data)):
    if class_list[j] not in temp.keys():
      temp[class_list[j]] = data[j]
    else:
      temp[class_list[j]] += data[j]
  return temp

def countWordPerClass(class_dict):
  word_dict = {}
  for classT in class_dict:
    words = set(class_dict[classT])
    for word in words:
      if word in word_dict.keys():
        word_dict[word] +=1
      else:
        word_dict[word] =1
  print(word_dict)
  return word_dict

def tf_Icf(words,class_dict,word_dict):
  count_words = Counter(words)
  tf_icf = {}
  for word in set(words):
    tf_value = count_words[word]
    icf_value = np.log(len(class_dict)/word_dict[word])
    tf_icf[word] = tf_value*icf_value
  print(tf_icf)
  return tf_icf
```

Data is split in the mentioned ratio and fit method is applied on the training data:

```
def fit(data):
  temp = createClassBasedWords(data[0].tolist(),data[0].tolist())
  words_list = []
  for i in temp:
    words_list += temp[i]
  word_dict = countWordPerClass(temp)
  tf_icf = tf_Icf(words_list,temp,word_dict)
  tf_icf_sorted = sorted(tf_icf.items(), key=lambda t: t[1],reverse=True)
  unique_words = [word_tf_icf[0] for word_tf_icf in tf_icf_sorted[:int(len(tf_icf_sorted)*10/100)]]
  wordFreCl= {}
  numWorCl={}
```

```
ratio_list = [0.5,0.7,0.8]
tmp_list = []
for ratio in ratio_list:
  data_train = df_docs.sample(frac=ratio,random_state=42)
  data_test = df_docs[~df_docs.index.isin(data_train.index)]
  fit(data_train)
```

Assumptions: No Assumptions