CSE508: Information Retrieval

Assignment 2

Rishita Chauhan
Zaid Ali
Harshita Gupta

1.

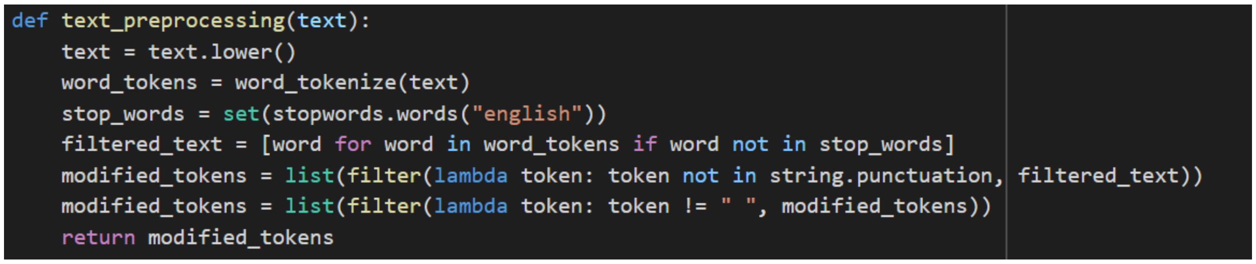
Preprocessing:

In this question, we extracted the relevant text and then performed the preprocessing steps:

- Lowercase the text
- Perform tokenization
- Remove stopwords
- Remove punctuations
- Remove blank space tokens

In the **text_preprocessing()** function, The input to the function is a string of text, which is first converted to lowercase using the lower() function. The text is then tokenized using the word_tokenize() function using the nltk package.
After that, a set of english stop words is created using the stopwords module from the nltk package, and the function removes all the stop words from the tokenized text using a list comprehension. The function then applies two additional filters using lambda functions to remove punctuation marks and spaces from the tokenized text.
Finally, the function returns a list of the modified tokens which are further used.

The returned value of this function is then saved for further use.



```python
def text_preprocessing(text):
    text = text.lower()
    word_tokens = word_tokenize(text)
    stop_words = set(stopwords.words("english"))
    filtered_text = [word for word in word_tokens if word not in stop_words]
    modified_tokens = list(filter(lambda token: token not in string.punctuation, filtered_text))
    modified_tokens = list(filter(lambda token: token != " ", modified_tokens))
    return modified_tokens
```

The result before and after preprocessing the data can be seen below:
Before:
RAW TEXT: experimental investigation of the aerodynamics of a wing in a slipstream .   an experimental study of a wing in a propeller slipstream was made in order to determine the spanwise

distribution of the lift increase due to slipstream at different angles of attack of the wing and at different free stream to slipstream velocity ratios . the results were intended in part as an evaluation basis for different theoretical treatments of this problem .   the comparative span loading curves, together with supporting evidence, showed that a substantial part of the lift increment produced by the slipstream was due to a /destalling/ or boundary-layer-control effect . the integrated remaining lift increment, after subtracting this destalling lift, was found to agree well with a potential flow theory .   an empirical evaluation of the destalling effects was made for the specific configuration of the experiment .

After:
LOWER CASED TEXT: experimental investigation of the aerodynamics of a wing in a slipstream .   an experimental study of a wing in a propeller slipstream was made in order to determine the spanwise distribution of the lift increase due to slipstream at different angles of attack of the wing and at different free stream to slipstream velocity ratios . the results were intended in part as an evaluation basis for different theoretical treatments of this problem .   the comparative span loading curves, together with supporting evidence, showed that a substantial part of the lift increment produced by the slipstream was due to a /destalling/ or boundary-layer-control effect . the integrated remaining lift increment, after subtracting this destalling lift, was found to agree well with a potential flow theory .   an empirical evaluation of the destalling effects was made for the specific configuration of the experiment .

TOKENIZED TEXT: ['experimental', 'investigation', 'of', 'the', 'aerodynamics', 'of', 'a', 'wing', 'in', 'a', 'slipstream', '.', 'an', 'experimental', 'study', 'of', 'a', 'wing', 'in', 'a', 'propeller', 'slipstream', 'was', 'made', 'in', 'order', 'to', 'determine', 'the', 'spanwise', 'distribution', 'of', 'the', 'lift', 'increase', 'due', 'to', 'slipstream', 'at', 'different', 'angles', 'of', 'attack', 'of', 'the', 'wing', 'and', 'at', 'different', 'free', 'stream', 'to', 'slipstream', 'velocity', 'ratios', '.', 'the', 'results', 'were', 'intended', 'in', 'part', 'as', 'an', 'evaluation', 'basis', 'for', 'different', 'theoretical', 'treatments', 'of', 'this', 'problem', '.', 'the', 'comparative', 'span', 'loading', 'curves', ',', 'together', 'with', 'supporting', 'evidence', ',', 'showed', 'that', 'a', 'substantial', 'part', 'of', 'the', 'lift', 'increment', 'produced', 'by', 'the', 'slipstream', 'was', 'due', 'to', 'a', '/destalling/', 'or', 'boundary-layer-control', 'effect', '.', 'the', 'integrated', 'remaining', 'lift', 'increment', ',', 'after', 'subtracting', 'this', 'destalling', 'lift', ',', 'was', 'found', 'to', 'agree', 'well', 'with', 'a', 'potential', 'flow', 'theory', '.', 'an', 'empirical', 'evaluation', 'of', 'the', 'destalling', 'effects', 'was', 'made', 'for', 'the', 'specific', 'configuration', 'of', 'the', 'experiment', '.']

STOPWORDS REMOVED: ['experimental', 'investigation', 'aerodynamics', 'wing', 'slipstream', '.', 'experimental', 'study', 'wing', 'propeller', 'slipstream', 'made', 'order', 'determine', 'spanwise', 'distribution', 'lift', 'increase', 'due', 'slipstream', 'different', 'angles', 'attack', 'wing', 'different', 'free', 'stream', 'slipstream', 'velocity', 'ratios', '.', 'results', 'intended', 'part', 'evaluation', 'basis', 'different', 'theoretical', 'treatments', 'problem', '.', 'comparative', 'span', 'loading', 'curves', ',', 'together', 'supporting', 'evidence', ',', 'showed', 'substantial', 'part', 'lift', 'increment', 'produced', 'slipstream', 'due', '/destalling/', 'boundary-layer-control', 'effect', '.', 'integrated', 'remaining', 'lift', 'increment', ',', 'subtracting', 'destalling', 'lift', ',', 'found', 'agree', 'well', 'potential', 'flow', 'theory', '.', 'empirical', 'evaluation', 'destalling', 'effects', 'made', 'specific', 'configuration', 'experiment', '.']

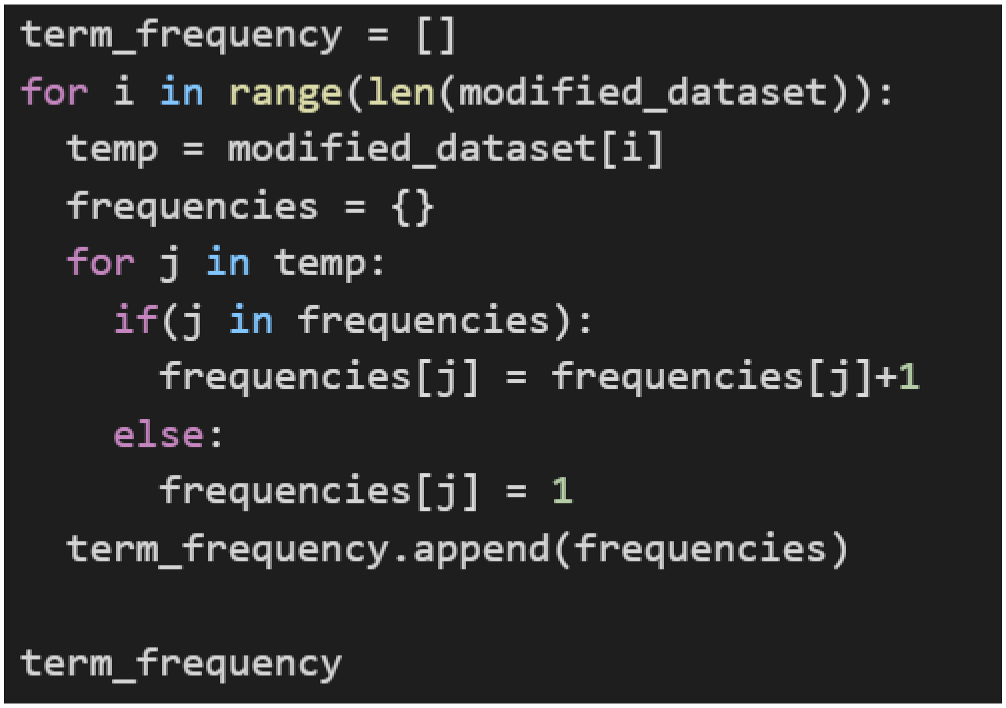
PUNCTUATION REMOVED: ['experimental', 'investigation', 'aerodynamics', 'wing', 'slipstream', 'experimental', 'study', 'wing', 'propeller', 'slipstream', 'made', 'order', 'determine', 'spanwise',

```
'distribution', 'lift', 'increase', 'due', 'slipstream', 'different', 'angles', 'attack', 'wing', 'different', 'free',
'stream', 'slipstream', 'velocity', 'ratios', 'results', 'intended', 'part', 'evaluation', 'basis', 'different',
'theoretical', 'treatments', 'problem', 'comparative', 'span', 'loading', 'curves', 'together', 'supporting',
'evidence', 'showed', 'substantial', 'part', 'lift', 'increment', 'produced', 'slipstream', 'due', '/destalling/',
'boundary-layer-control', 'effect', 'integrated', 'remaining', 'lift', 'increment', 'subtracting', 'destalling',
'lift', 'found', 'agree', 'well', 'potential', 'flow', 'theory', 'empirical', 'evaluation', 'destalling', 'effects',
'made', 'specific', 'configuration', 'experiment']

BLANK SPACE TOKENS REMOVED:  ['experimental', 'investigation', 'aerodynamics', 'wing',
'slipstream', 'experimental', 'study', 'wing', 'propeller', 'slipstream', 'made', 'order', 'determine',
'spanwise', 'distribution', 'lift', 'increase', 'due', 'slipstream', 'different', 'angles', 'attack', 'wing',
'different', 'free', 'stream', 'slipstream', 'velocity', 'ratios', 'results', 'intended', 'part', 'evaluation', 'basis',
'different', 'theoretical', 'treatments', 'problem', 'comparative', 'span', 'loading', 'curves', 'together',
'supporting', 'evidence', 'showed', 'substantial', 'part', 'lift', 'increment', 'produced', 'slipstream', 'due',
'/destalling/', 'boundary-layer-control', 'effect', 'integrated', 'remaining', 'lift', 'increment', 'subtracting',
'destalling', 'lift', 'found', 'agree', 'well', 'potential', 'flow', 'theory', 'empirical', 'evaluation', 'destalling',
'effects', 'made', 'specific', 'configuration', 'experiment']
```

We then made the TF-IDF matrix for every term in the vocabulary and determined the TF-IDF score for a given query. The TF-IDF consists of two components - Term Frequency and Inverse Document Frequency.

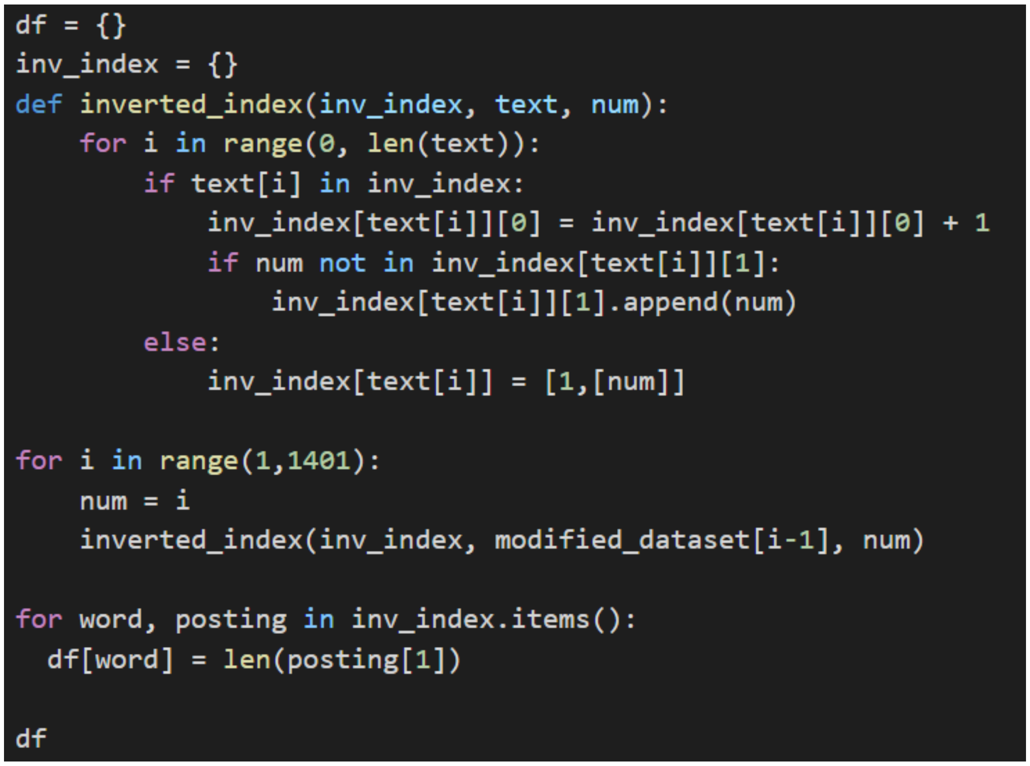
- To compute the Term Frequency, the frequency count of each term in every document is computed and stored as a nested dictionary for each document.
- To determine the document frequency of each term, the postings list of each term is found and the number of documents in each posting list is counted.
- The IDF value of each term is calculated using the following formula with smoothing: IDF(term) = log(total number of documents/document frequency(term)+1).
- The Term Frequency is calculated using 5 different weighting schemes, as listed below:

The term frequencies were calculated as follows:



```
term_frequency = []
for i in range(len(modified_dataset)):
  temp = modified_dataset[i]
  frequencies = {}
  for j in temp:
    if(j in frequencies):
      frequencies[j] = frequencies[j]+1
    else:
      frequencies[j] = 1
  term_frequency.append(frequencies)

term_frequency
```

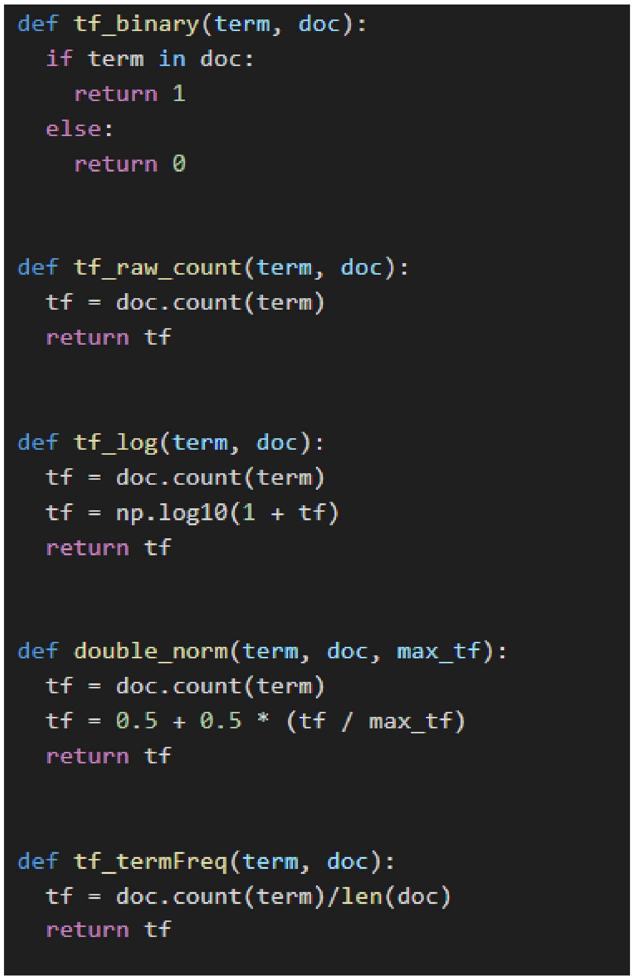
Document frequency was calculated as follows:



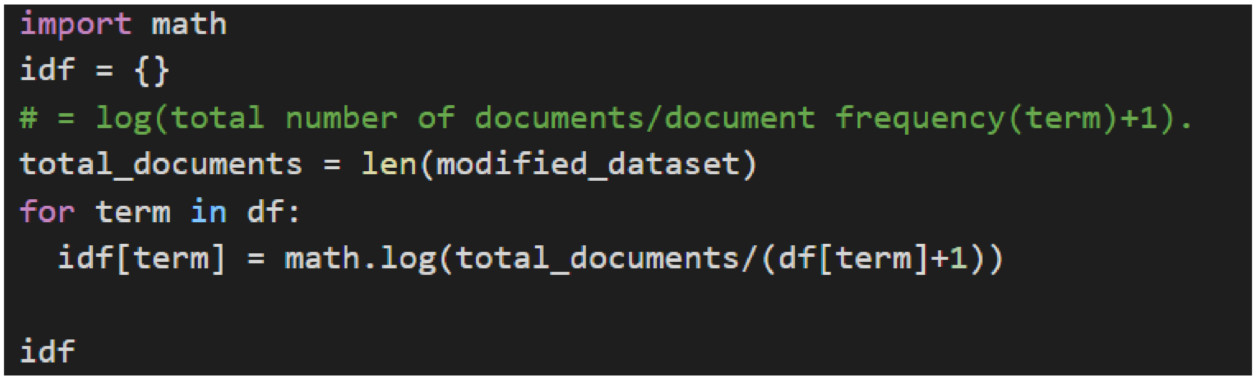
```
df = {}
inv_index = {}
def inverted_index(inv_index, text, num):
    for i in range(0, len(text)):
        if text[i] in inv_index:
            inv_index[text[i]][0] = inv_index[text[i]][0] + 1
            if num not in inv_index[text[i]][1]:
                inv_index[text[i]][1].append(num)
        else:
            inv_index[text[i]] = [1,[num]]

for i in range(1,1401):
    num = i
    inverted_index(inv_index, modified_dataset[i-1], num)

for word, posting in inv_index.items():
  df[word] = len(posting[1])

df
```

The above code constructs an inverted index dictionary and calculates the document frequency (df) for each term in a collection of text documents.
We first initialize an empty dictionary inv_index to store the inverted index and an empty dictionary df to store the document frequency of each term. After iterating over a range of 1 to 1400 inverted_index function is called for each document. The inverted_index function takes
three arguments: the inv_index dictionary, the text of the document as a list of preprocessed tokens (modified_dataset[i-1]), and the document number (num). The function updates the inv_index dictionary by adding each term in the document to its postings list. If the term is already in the dictionary, the inv_index function increments its document frequency count and adds the current document number to its postings list. If the term is not already in the dictionary, the function adds it to the dictionary with a document frequency count of 1 and initializes its postings list with the current document number.
After constructing the inverted index, The document frequency is calculated by iterating over the inv_index dictionary for each term by counting the number of unique document numbers in its postings list. Document frequency is stored in the df dictionary with the term as key and the document frequency as the value.

We first got the entire corpus and then implemented the 5 variants of Term Frequency.


```
def tf_binary(term, doc):
  if term in doc:
    return 1
  else:
    return 0


def tf_raw_count(term, doc):
  tf = doc.count(term)
  return tf


def tf_log(term, doc):
  tf = doc.count(term)
  tf = np.log10(1 + tf)
  return tf


def double_norm(term, doc, max_tf):
  tf = doc.count(term)
  tf = 0.5 + 0.5 * (tf / max_tf)
  return tf


def tf_termFreq(term, doc):
  tf = doc.count(term)/len(doc)
  return tf
```

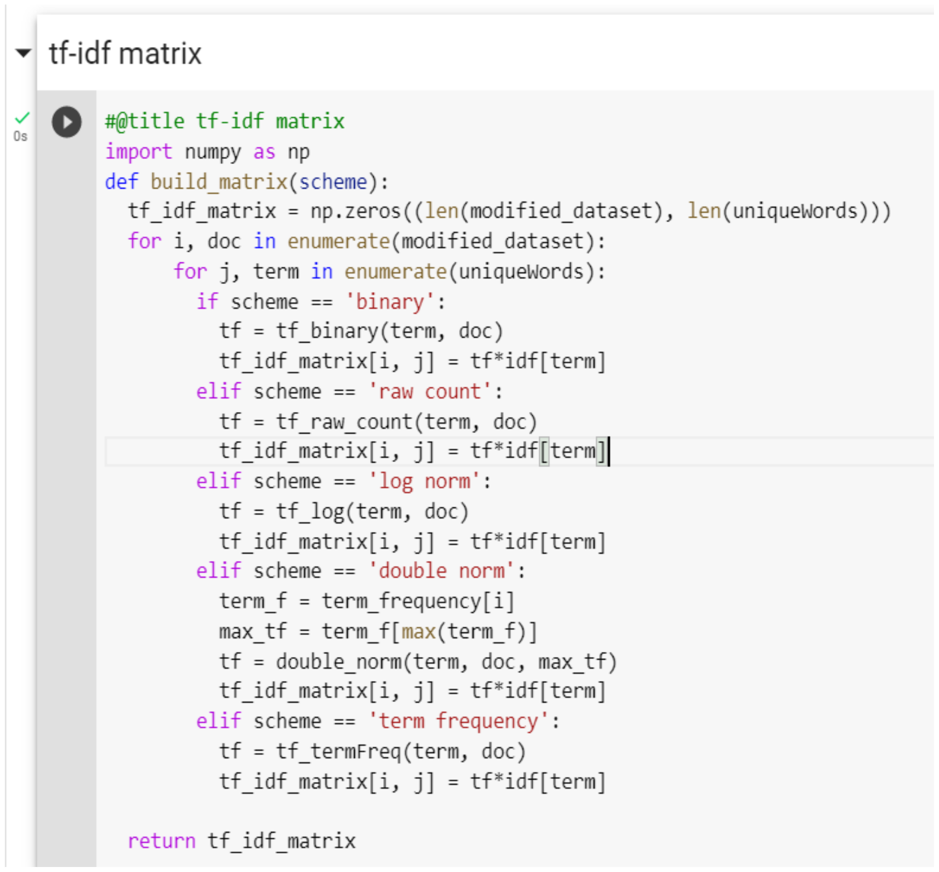
IDF:



```python
import math
idf = {}
# = log(total number of documents/document frequency(term)+1).
total_documents = len(modified_dataset)
for term in df:
  idf[term] = math.log(total_documents/(df[term]+1))

idf
```

An empty dictionary(idf) is initialized to store inverse document frequency for each term in the corpus. Idf is calculated by iterating over the df dictionary. For each term in the df dictionary idf is calculated using the formula: idf = log(total number of documents/document frequency(term)+1). The calculated value is stored in the idf dictionary.

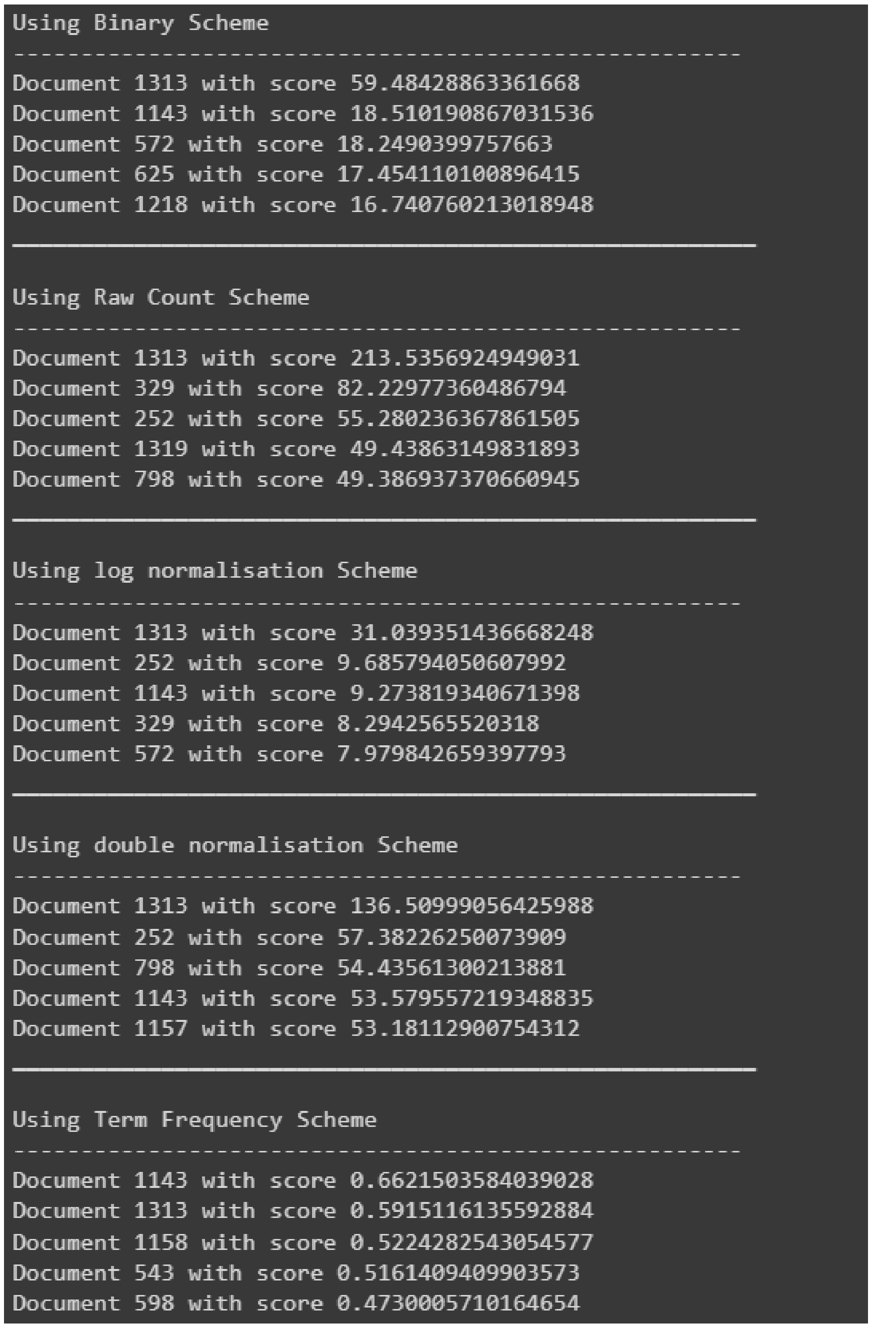
TF-IDF:



```python
#@title tf-idf matrix
import numpy as np
def build_matrix(scheme):
  tf_idf_matrix = np.zeros((len(modified_dataset), len(uniqueWords)))
  for i, doc in enumerate(modified_dataset):
      for j, term in enumerate(uniqueWords):
        if scheme == 'binary':
          tf = tf_binary(term, doc)
          tf_idf_matrix[i, j] = tf*idf[term]
        elif scheme == 'raw count':
          tf = tf_raw_count(term, doc)
          tf_idf_matrix[i, j] = tf*idf[term]
        elif scheme == 'log norm':
          tf = tf_log(term, doc)
          tf_idf_matrix[i, j] = tf*idf[term]
        elif scheme == 'double norm':
          term_f = term_frequency[i]
          max_tf = term_f[max(term_f)]
          tf = double_norm(term, doc, max_tf)
          tf_idf_matrix[i, j] = tf*idf[term]
        elif scheme == 'term frequency':
          tf = tf_termFreq(term, doc)
          tf_idf_matrix[i, j] = tf*idf[term]

  return tf_idf_matrix
```

After calculating the tf and idf values for each term in the corpus. We finally calculating the tf-idf matrix by using the following formula: tf-idf = tf*idf[term]. The build matrix function takes the scheme as an argument. tf_idf_matrix is initialized for storing tf_idf values. Further the tf-idf values are calculated with respect to the given scheme and are stored in the tf_idf_matrix. The tf_idf_matrix is returned by the function.

Query vector and score of the top 5 relevant documents is calculated and reported as follows:



```
Using Binary Scheme
------------------------------------------------------
Document 1313 with score 59.48428863361668
Document 1143 with score 18.510190867031536
Document 572 with score 18.2490399757663
Document 625 with score 17.454110100896415
Document 1218 with score 16.740760213018948
______________________________________________________

Using Raw Count Scheme
------------------------------------------------------
Document 1313 with score 213.5356924949031
Document 329 with score 82.22977360486794
Document 252 with score 55.280236367861505
Document 1319 with score 49.43863149831893
Document 798 with score 49.386937370660945
______________________________________________________

Using log normalisation Scheme
------------------------------------------------------
Document 1313 with score 31.039351436668248
Document 252 with score 9.685794050607992
Document 1143 with score 9.273819340671398
Document 329 with score 8.2942565520318
Document 572 with score 7.979842659397793
______________________________________________________

Using double normalisation Scheme
------------------------------------------------------
Document 1313 with score 136.50999056425988
Document 252 with score 57.38226250073909
Document 798 with score 54.43561300213881
Document 1143 with score 53.579557219348835
Document 1157 with score 53.18112900754312
______________________________________________________

Using Term Frequency Scheme
------------------------------------------------------
Document 1143 with score 0.6621503584039028
Document 1313 with score 0.5915116135592884
Document 1158 with score 0.5224282543054577
Document 543 with score 0.5161409409903573
Document 598 with score 0.4730005710164654
```

PROS AND CONS OF EACH WEIGHTING SCHEMES:

1. **Binary:** Using a binary weighting scheme, the presence or absence of a term in a document is represented as a binary value (0 or 1), with 1 denoting the term's presence and 0 denoting its absence. One benefit of using binary weighting scheme is that it is straightforward and simple to use. It is computationally efficient because it simply needs to examine whether a phrase is present or absent in a document. Although, the binary weighting method disregards how frequently a term appears in a document which becomes a problem when searching for documents that contain rare terms. The relevance of a document to a query might not be captured by the binary weighting technique. Even though the latter is more relevant, a document with just one instance of a query phrase may be given the same ranking as one with several instances.

2. **Raw Count:** Raw count weighting scheme involves representing the frequency of occurrence of a term in a document as a raw count. It captures the frequency with which a term appears in a text, which is significant when looking for papers that include particular or uncommon terms.The raw count weighting system is simple to comprehend and intuitive. It is quite easy to understand that a document is more likely to be relevant if it has a higher raw count for a specific term. But at the same time the raw count weighting scheme is length-sensitive, that is, longer documents may have higher raw counts for terms even if they are not more important.

3. **Log normalization:** Log normalization weighting scheme involves taking the logarithm of the raw frequency of a term in a document. The term frequency is normalised to a logarithmic scale during log normalisation, which lessens the effect of long texts on term frequency weighting. By decreasing the weight of extremely frequent phrases and raising the weight of less frequent terms, log normalisation achieves a balance between terms with high and low frequency. But logarithm of the raw term frequency must be calculated by additional processing during log normalisation, which could complicate the computation. As frequency is converted to a logarithmic scale during log normalisation, precise frequency information about terms may be lost.

4. **Double normalization:** By normalising the term frequency in relation to the maximum frequency of any word in the same document, double normalisation lessens the influence of document length on term frequency weighting. It helps in capturing rare terms by increasing their weight relative to more common terms. By dividing the term frequency by the overall number of instances of the term across all texts, double normalisation balances terms with high and low frequency. However, the computational complexity of double normalisation may increase since it necessitates additional processing to determine the maximum term frequency in a document and the total number of term occurrences across all documents. Also, if the highest frequency of any phrase in a document is significantly higher than the frequency of the term being weighted, double normalisation may lead to skewed weighting.

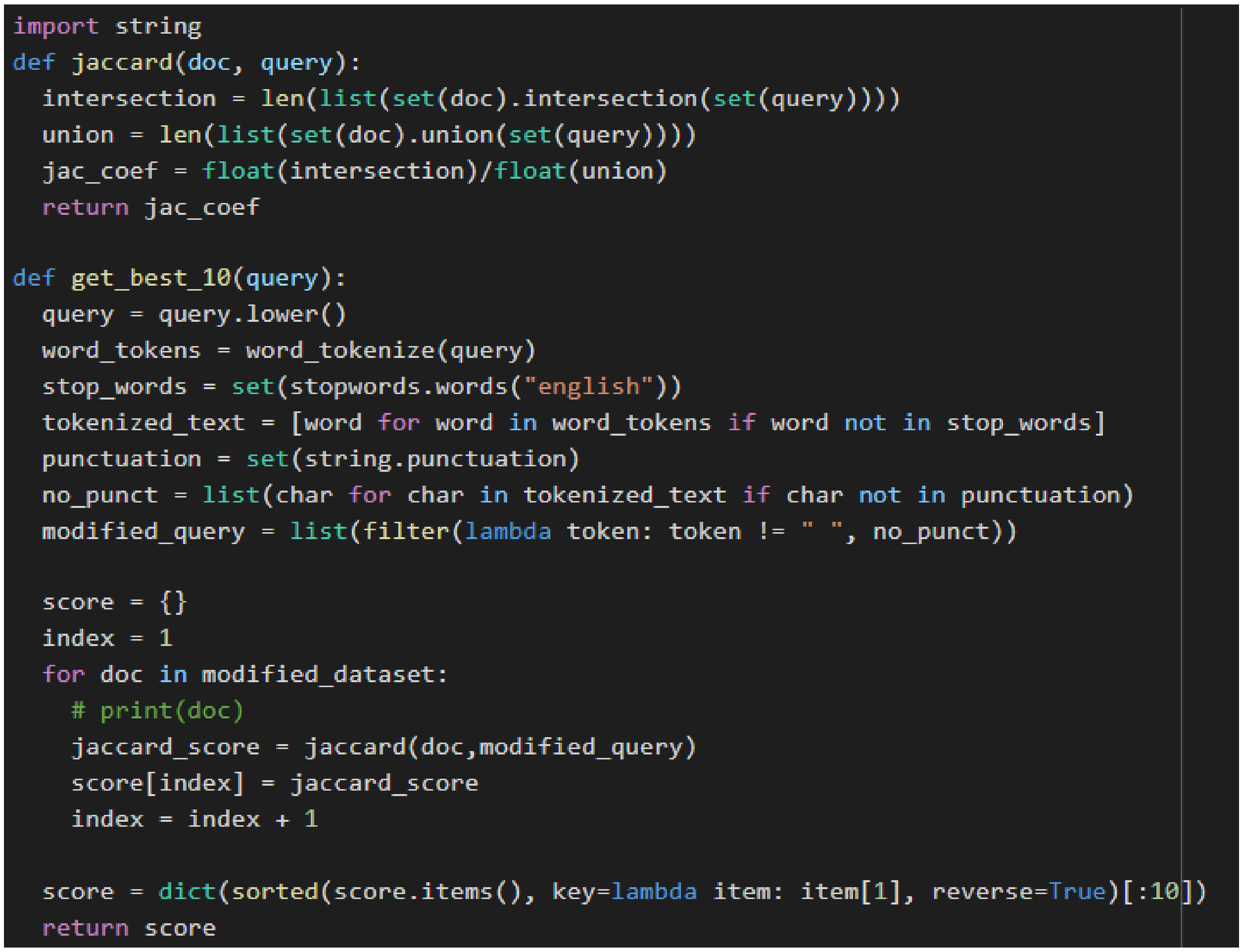
5. **Term Frequency:** It entails figuring out how frequently a term appears in a document and utilising that frequency as the term's weight. By giving more weight to terms that appear more frequently than others, term frequency weighting captures the significance of terms in a document. One of the disadvantages of term frequency weighting is that high frequency terms

have the potential to distort term frequency weighting and make it challenging to distinguish between documents. It also ignores the length of the document, which can be problematic for lengthy documents where the frequency of a term may not appropriately reflect its significance.

PART 2: JACCARD COEFFICIENT

Then we had to calculate the Jaccard Coefficient. A higher Jaccard coefficient value indicates greater relevance of the document to the query.

Jaccard Coefficient = Intersection of (doc,query) / Union of (doc,query).



```python
import string
def jaccard(doc, query):
  intersection = len(list(set(doc).intersection(set(query))))
  union = len(list(set(doc).union(set(query))))
  jac_coef = float(intersection)/float(union)
  return jac_coef

def get_best_10(query):
  query = query.lower()
  word_tokens = word_tokenize(query)
  stop_words = set(stopwords.words("english"))
  tokenized_text = [word for word in word_tokens if word not in stop_words]
  punctuation = set(string.punctuation)
  no_punct = list(char for char in tokenized_text if char not in punctuation)
  modified_query = list(filter(lambda token: token != " ", no_punct))

  score = {}
  index = 1
  for doc in modified_dataset:
    # print(doc)
    jaccard_score = jaccard(doc,modified_query)
    score[index] = jaccard_score
    index = index + 1

  score = dict(sorted(score.items(), key=lambda item: item[1], reverse=True)[:10])
  return score
```

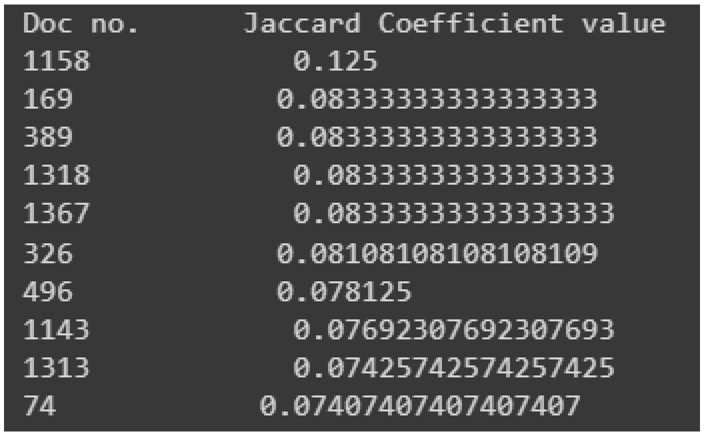
We defined two functions **jaccard** and **get_best_10** to calculate the Jaccard coefficient and return the top 10 documents that are most similar to the input query, respectively.

The jaccard function takes two input parameters, doc and query, and returns the Jaccard coefficient between them. The jaccard coefficient is calculated as:

jacc_coef: **(length of intersection)/(length of union)**

The **get_best_10** function takes a string query as input and tokenizes it into words, removes stop words and punctuation, and returns a modified query list. Then, it calculates the Jaccard coefficient between the modified query and each document. Finally, it returns the top 10 documents ranked by their Jaccard coefficient value as a dictionary where the key is the document number and the value is its Jaccard coefficient.

Finally we find the top 10 documents that are most similar to the given query and print the document number and its corresponding jaccard coefficient.

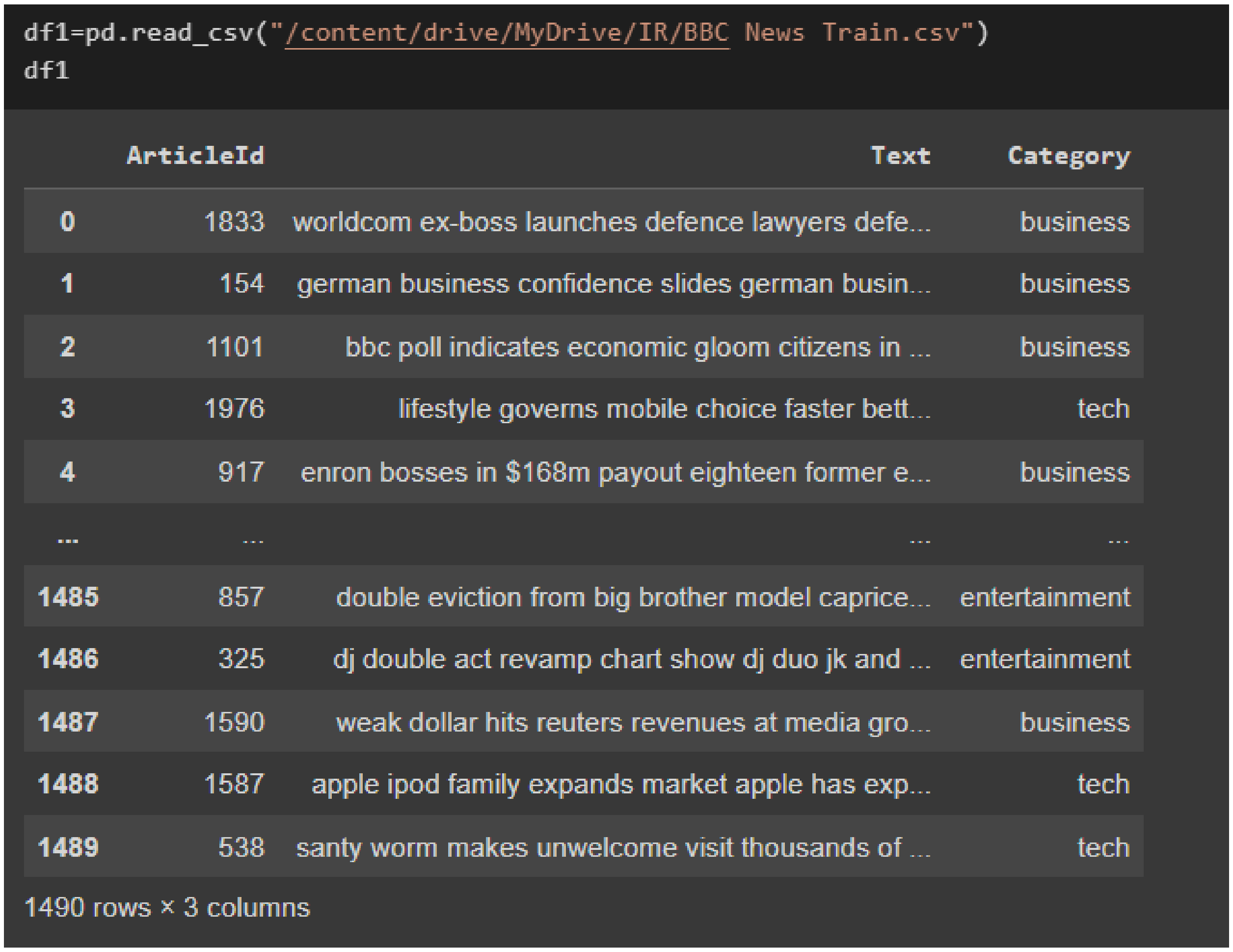


```
Doc no.      Jaccard Coefficient value
1158           0.125
169           0.08333333333333333
389           0.08333333333333333
1318           0.08333333333333333
1367           0.08333333333333333
326           0.08108108108108109
496           0.078125
1143           0.07692307692307693
1313           0.07425742574257425
74           0.07407407407407407
```

2.

In this question, we used the BBC News Classification dataset. We downloaded the dataset and performed the preprocessing steps. The dataset had three columns - article id, text, category.

Original dataset:



```
df1=pd.read_csv("/content/drive/MyDrive/IR/BBC News Train.csv")
df1
```

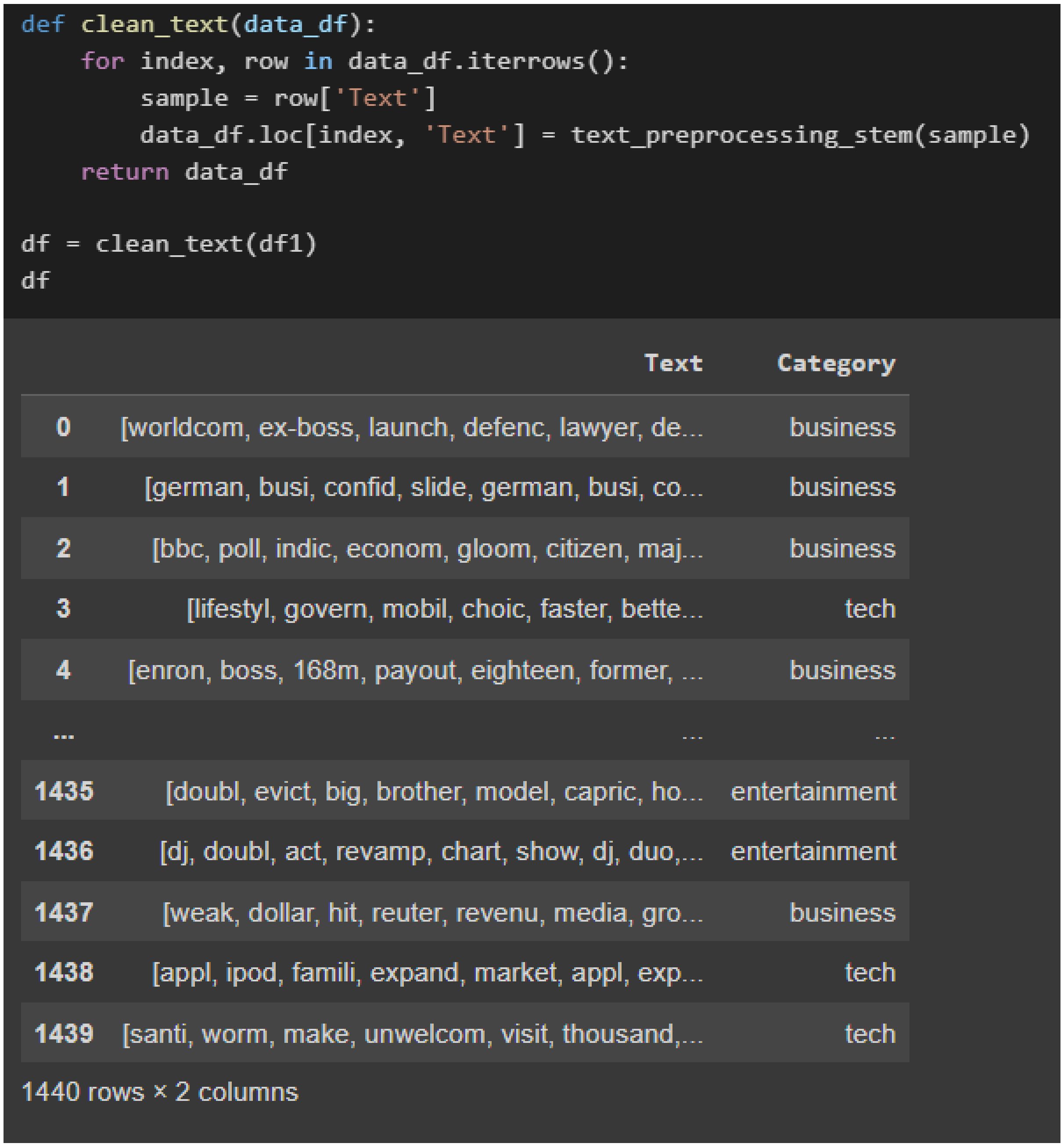
	ArticleId	Text	Category
0	1833	worldcom ex-boss launches defence lawyers defe...	business
1	154	german business confidence slides german busin...	business
2	1101	bbc poll indicates economic gloom citizens in ...	business
3	1976	lifestyle governs mobile choice faster bett...	tech
4	917	enron bosses in $168m payout eighteen former e...	business
...	...	...	...
1485	857	double eviction from big brother model caprice...	entertainment
1486	325	dj double act revamp chart show dj duo jk and ...	entertainment
1487	1590	weak dollar hits reuters revenues at media gro...	business
1488	1587	apple ipod family expands market apple has exp...	tech
1489	538	santy worm makes unwelcome visit thousands of ...	tech

1490 rows × 3 columns

We dropped the unnecessary column - Article Id. We also dropped the duplicate values from the dataset.

Further the dataset was cleaned by removing the stop words, punctuations, converting the text to lower case, and tokenizing the text. We then performed stemming to reduce words to their root words.

Preprocessed dataset:



```python
def clean_text(data_df):
    for index, row in data_df.iterrows():
        sample = row['Text']
        data_df.loc[index, 'Text'] = text_preprocessing_stem(sample)
    return data_df

df = clean_text(df1)
df
```

```
                                                Text       Category
0     [worldcom, ex-boss, launch, defenc, lawyer, de...      business
1        [german, busi, confid, slide, german, busi, co...   business
2        [bbc, poll, indic, econom, gloom, citizen, maj...   business
3           [lifestyl, govern, mobil, choic, faster, bette...    tech
4     [enron, boss, 168m, payout, eighteen, former, ...      business
...                                                  ...          ...
1435     [doubl, evict, big, brother, model, capric, ho...  entertainment
1436     [dj, doubl, act, revamp, chart, show, dj, duo,...  entertainment
1437     [weak, dollar, hit, reuter, revenu, media, gro...   business
1438     [appl, ipod, famili, expand, market, appl, exp...       tech
1439  [santi, worm, make, unwelcom, visit, thousand,...         tech
1440 rows × 2 columns
```

The resultant preprocessed documents are grouped by their classes and their texts are combined.

We then implemented the TF-ICF on the preprocessed dataset. TF-ICF stands for **term frequency-inverse category frequency.** It is a weighting scheme that is commonly used in information retrieval and text classification tasks to assign weights to terms in a particular class.

Formula for calculating TF-ICF:

TF-ICF = TF * log(N/CF)

Here,
TF: Number of occurrences of a term in all documents of a particular class
N: Number of classes
CF : Number of classes in which that term occurs

```
def tf(word, postings, class_id, class_doc):
    tf = postings[word][1][class_id]
    return tf

def tf_log(word, postings, class_id, class_doc):
    tf = postings[word][1][class_id]
    tf = math.log(1 + tf)
    return tf

def tf_binary(word, postings, class_id, class_doc):
  if class_id in postings[word][1]:
    return 1
  else:
    return 0

def icf(word, postings):
    num_classes = 5
    return math.log(num_classes / len(postings[word][1]))

def tficf(word, postings, class_id, class_doc, scheme = None):
    if scheme == None:
      return tf(word, postings, class_id, class_doc) * icf(word, postings)
    elif scheme == 'log':
      return tf_log(word, postings, class_id, class_doc) * icf(word,
postings)
    elif scheme == 'binary':
      return tf_binary(word, postings, class_id, class_doc) * icf(word,
postings)

def posting_list(df):
    postings = {}

    for index, row in df.iterrows():
        for token in row['Text']:
            if token in postings:
                if row['Category'] in postings[token][1]:
                  postings[token][1][row['Category']] =
postings[token][1][row['Category']] + 1
                else:
                  postings[token][1][row['Category']] =  1
                postings[token][0] = 1 + postings[token][0]
            else:
                postings[token] = [1, {row['Category']:1}]

    return postings

def build_tc_icf_mat(postings, category_club, scheme = None):
  num_classes = ['business', 'entertainment', 'politics', 'sport', 'tech']
  tf_icf_matrix = {}
  posting_list = postings.items()
  for i in num_classes:
    tf_icf_matrix[i] = {}
    for j, term in enumerate(posting_list):
          term = term[0]
          tf_icf_matrix[i][term] = 0
          try:
            tf_icf_matrix[i][term] = tficf(term, postings, i,
category_club[category_club['Category']==i]['Text'].values[0], scheme)
          except:
            pass
  return tf_icf_matrix
```

Implementation of Naive Baye's:

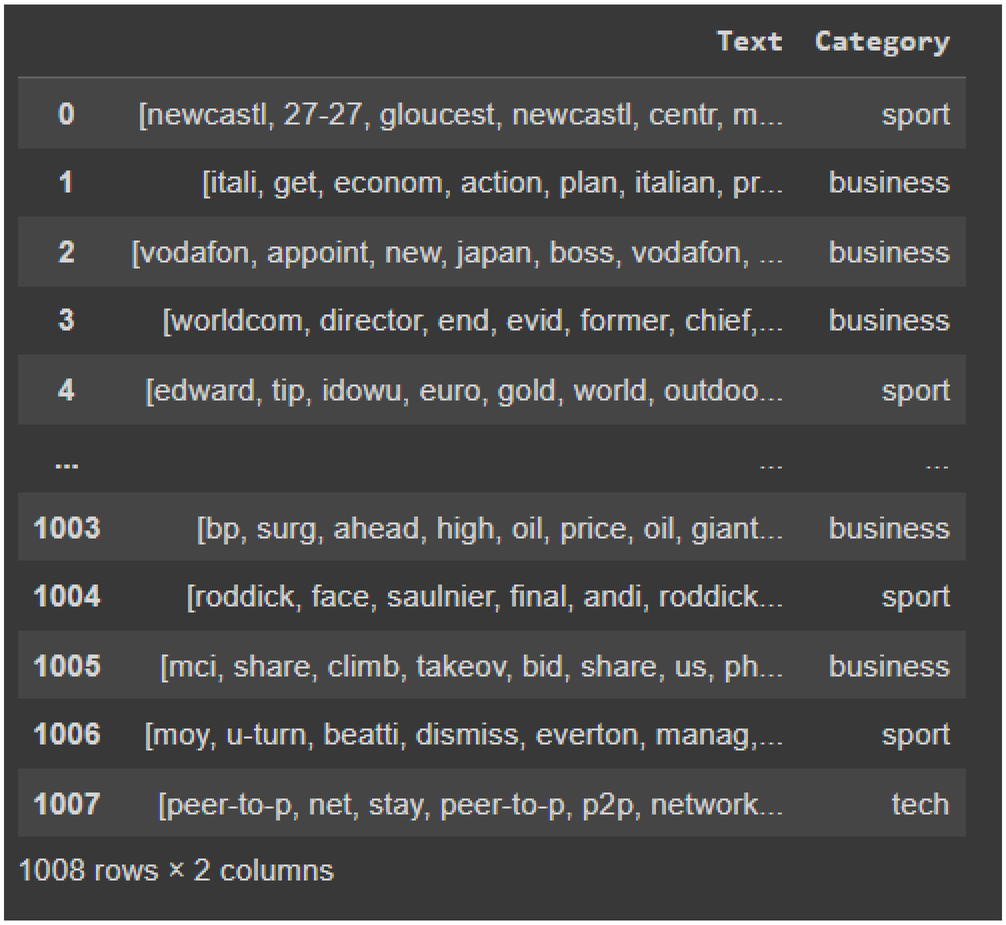
```
def get_class_prob(df_train):
  prob_cat = {}
  val_count = df_train['Category'].value_counts()
  for i in val_count.index:
    prob_cat[i] = val_count[i]/val_count.sum()
  return prob_cat
```

```python
def train(df_train, scheme = None):
  category_club = 
df_train.groupby('Category')['Text'].agg(sum).reset_index()
  postings = posting_list(category_club)
  tf_icf_matrix = build_tc_icf_mat(postings, category_club, scheme)
  prob_cat = get_class_prob(df_train)
  return tf_icf_matrix, prob_cat

def predict(X, tf_icf_matrix, prob_cat):
  num_classes = ['business', 'entertainment', 'politics', 'sport', 'tech']
  y_pred = []
  for i in X:
    prob_a = {}
    for cat in num_classes:
      prior = math.log(prob_cat[cat])
      for word in i:
        if word in tf_icf_matrix[cat]:
          prior = prior +  tf_icf_matrix[cat][word]
        else:
          prior = prior + math.log(1/(len(X)+1))
      prob_a[cat] = prior
    y_pred.append(max(prob_a, key= lambda x: prob_a[x]))
  return y_pred
```

Testing the model on our dataset, when the train test ratio is 70:30:

```python
split = 0.3
X_train, X_test, y_train, y_test = train_test_split(df['Text'],
df['Category'], test_size=split, random_state=42, stratify=df['Category'])
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

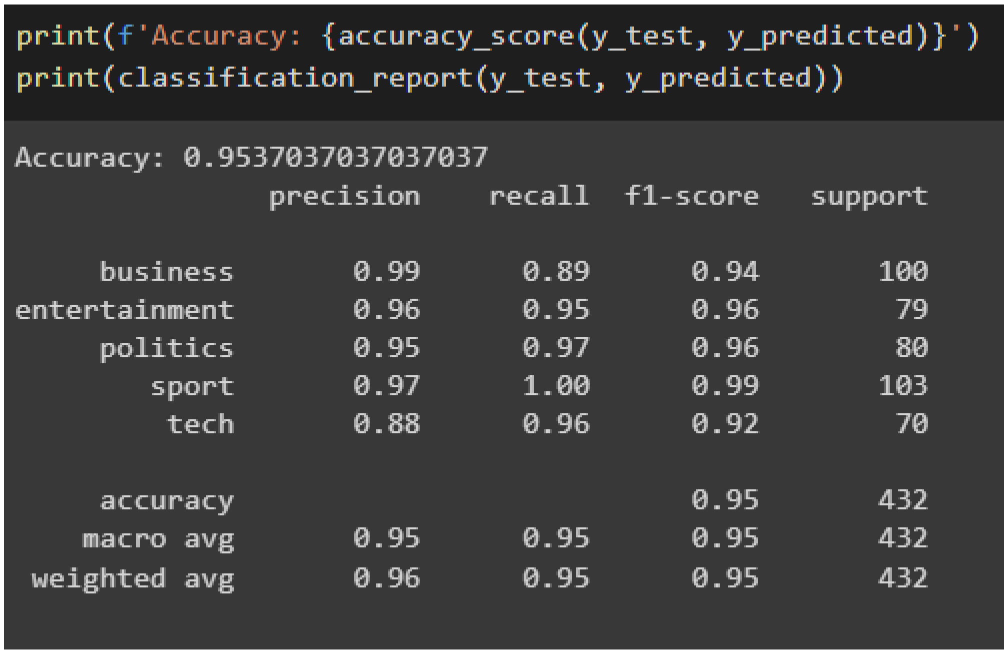
```python
data = {"Text": X_train,
        "Category": y_train}
df_train = pd.concat(data, axis = 1).reset_index(drop=True)
df_train
```



	Text	Category
0	[newcastl, 27-27, gloucest, newcastl, centr, m...	sport
1	[itali, get, econom, action, plan, italian, pr...	business
2	[vodafon, appoint, new, japan, boss, vodafon, ...	business
3	[worldcom, director, end, evid, former, chief,...	business
4	[edward, tip, idowu, euro, gold, world, outdoo...	sport
...	...	...
1003	[bp, surg, ahead, high, oil, price, oil, giant...	business
1004	[roddick, face, saulnier, final, andi, roddick...	sport
1005	[mci, share, climb, takeov, bid, share, us, ph...	business
1006	[moy, u-turn, beatti, dismiss, everton, manag,...	sport
1007	[peer-to-p, net, stay, peer-to-p, p2p, network...	tech

1008 rows × 2 columns

Accuracy:



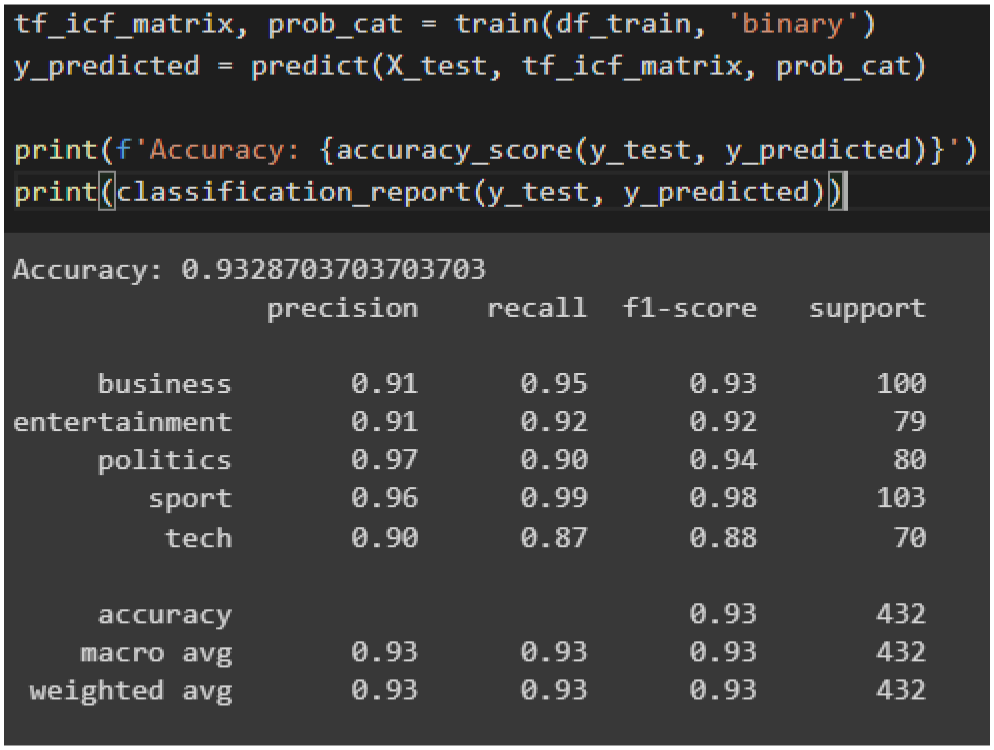
```
print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

```
Accuracy: 0.9537037037037037
               precision    recall  f1-score   support

     business       0.99      0.89      0.94       100
entertainment       0.96      0.95      0.96        79
     politics       0.95      0.97      0.96        80
        sport       0.97      1.00      0.99       103
         tech       0.88      0.96      0.92        70

     accuracy                           0.95       432
    macro avg       0.95      0.95      0.95       432
 weighted avg       0.96      0.95      0.95       432
```

<u>CONCLUSION AND FINDINGS:</u>

For improving the classifier we tried different weighting schemes:

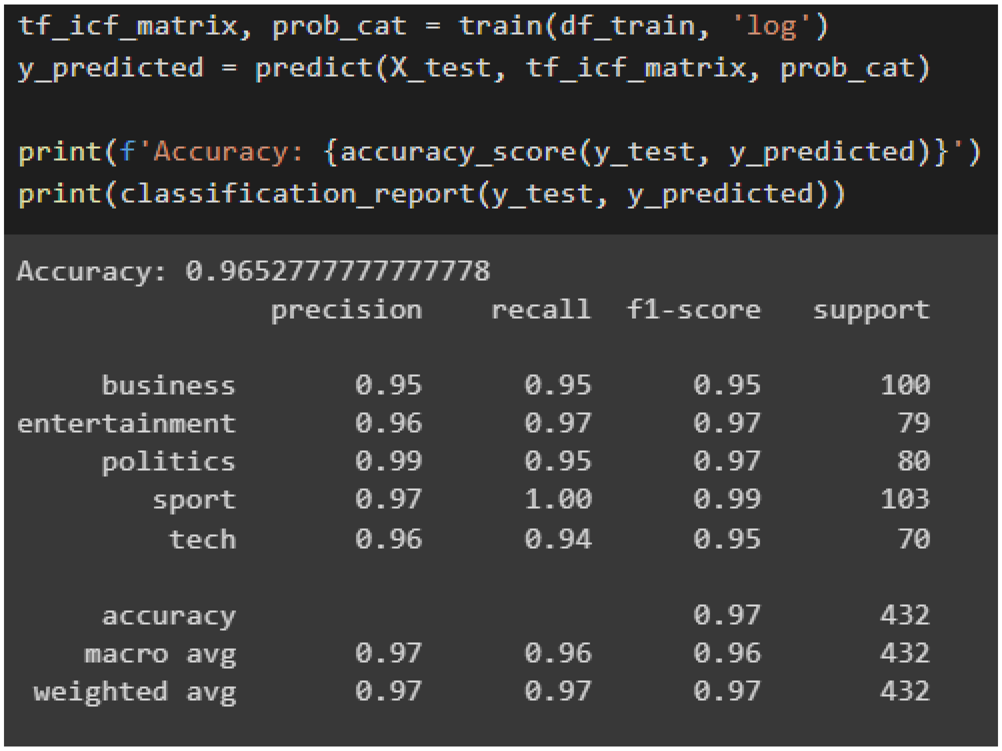
a. Binary weighting scheme:



```
tf_icf_matrix, prob_cat = train(df_train, 'binary')
y_predicted = predict(X_test, tf_icf_matrix, prob_cat)

print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

```
Accuracy: 0.9328703703703703
               precision    recall  f1-score   support

     business       0.91      0.95      0.93       100
entertainment       0.91      0.92      0.92        79
     politics       0.97      0.90      0.94        80
        sport       0.96      0.99      0.98       103
         tech       0.90      0.87      0.88        70

     accuracy                           0.93       432
    macro avg       0.93      0.93      0.93       432
 weighted avg       0.93      0.93      0.93       432
```

b. Log weighting scheme:



```
tf_icf_matrix, prob_cat = train(df_train, 'log')
y_predicted = predict(X_test, tf_icf_matrix, prob_cat)

print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

```
Accuracy: 0.9652777777777778
               precision    recall  f1-score   support

     business       0.95      0.95      0.95       100
entertainment       0.96      0.97      0.97        79
     politics       0.99      0.95      0.97        80
        sport       0.97      1.00      0.99       103
         tech       0.96      0.94      0.95        70

     accuracy                           0.97       432
    macro avg       0.97      0.96      0.96       432
 weighted avg       0.97      0.97      0.97       432
```

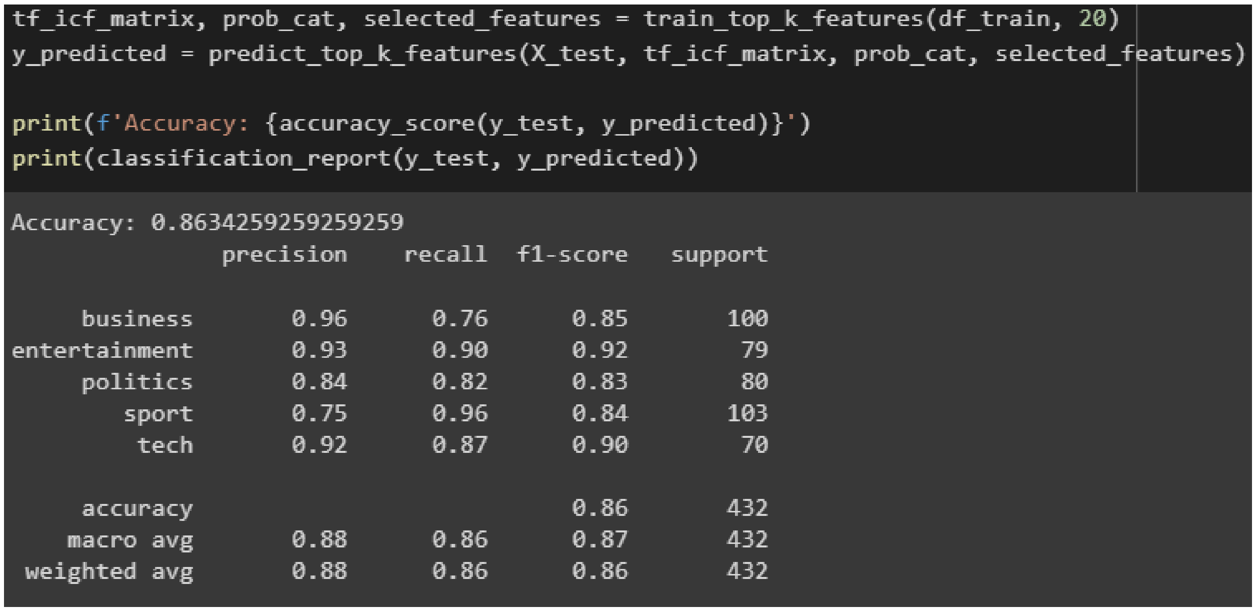
Inference:

It is evident that when we used the binary weighting scheme, the accuracy decreased as the binary weighting method disregards how frequently a term appears in a document which becomes a problem when searching for documents that contain rare terms. The relevance of a document to a query is not captured by the binary weighting technique. Even though the latter is more relevant, a document with just one instance of a query phrase may be given the same ranking as one with several instances.

But at the same time, using log weighting scheme improved the accuracy when compared to raw count weighting scheme as the term frequency is normalised to a logarithmic scale during log normalisation, which lessens the effect of long texts on term frequency weighting. By decreasing the weight of extremely frequent phrases and raising the weight of less frequent terms, log normalisation achieves a balance between terms with high and low frequency.

Result after using top k features:

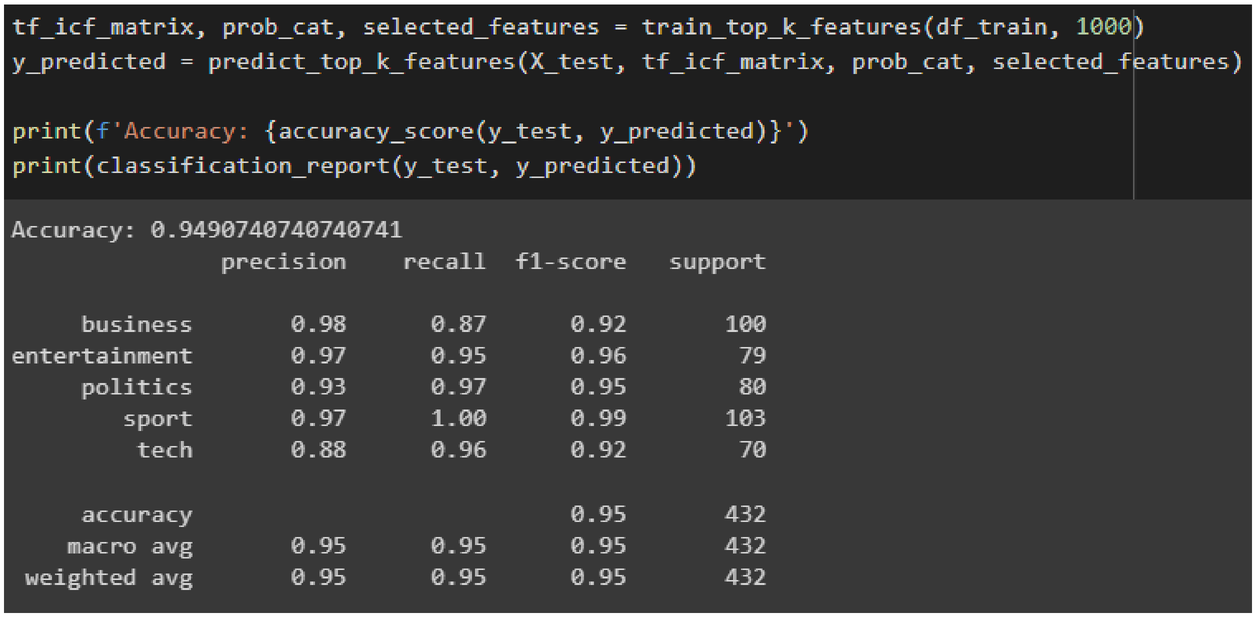
Top 20 features:



```
tf_icf_matrix, prob_cat, selected_features = train_top_k_features(df_train, 20)
y_predicted = predict_top_k_features(X_test, tf_icf_matrix, prob_cat, selected_features)

print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

```
Accuracy: 0.8634259259259259
               precision    recall  f1-score   support

     business       0.96      0.76      0.85       100
entertainment       0.93      0.90      0.92        79
     politics       0.84      0.82      0.83        80
        sport       0.75      0.96      0.84       103
         tech       0.92      0.87      0.90        70

     accuracy                           0.86       432
    macro avg       0.88      0.86      0.87       432
 weighted avg       0.88      0.86      0.86       432
```

Top 1000 features:



```
tf_icf_matrix, prob_cat, selected_features = train_top_k_features(df_train, 1000)
y_predicted = predict_top_k_features(X_test, tf_icf_matrix, prob_cat, selected_features)

print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

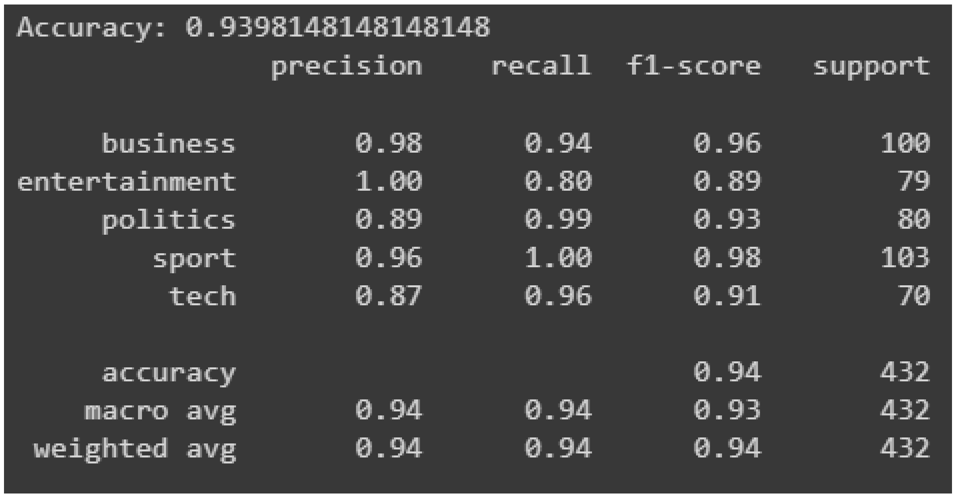
```
Accuracy: 0.9490740740740741
               precision    recall  f1-score   support

     business       0.98      0.87      0.92       100
entertainment       0.97      0.95      0.96        79
     politics       0.93      0.97      0.95        80
        sport       0.97      1.00      0.99       103
         tech       0.88      0.96      0.92        70

     accuracy                           0.95       432
    macro avg       0.95      0.95      0.95       432
 weighted avg       0.95      0.95      0.95       432
```

Inference

From the above results, we concluded that as we decreased the size of features the performance of the classifier decreased.

Result after using TF-IDF weighting scheme:

```
def idf(word, doc_postings, total_docs):
    return math.log(total_docs / len(doc_postings[word][1]))

def tfidf(word, postings, class_id, class_doc, doc_postings, total_docs):
    return tf(word, postings, class_id, class_doc) * idf(word,
doc_postings, total_docs)

def doc_posting_list(df):
    postings = {}
    for index, row in df.iterrows():
        for token in row['Text']:
            if token in postings:
                if index not in postings[token][1]:
                  postings[token][1].append(index)
                postings[token][0] = 1 + postings[token][0]
            else:
                postings[token] = [1, [index]]

    return postings

def build_tf_idf_mat(postings, category_club, doc_postings, total_docs):
  num_classes = ['business', 'entertainment', 'politics', 'sport', 'tech']
  tf_idf_matrix = {}
  posting_list = postings.items()
  for i in num_classes:
    tf_idf_matrix[i] = {}
    for j, term in enumerate(posting_list):
          term = term[0]
          tf_idf_matrix[i][term] = 0
          try:
            tf_idf_matrix[i][term] = tfidf(term, postings, i,
category_club[category_club['Category']==i]['Text'].values[0],
doc_postings, total_docs)
          except:
            pass
  return tf_idf_matrix

def train_tf_idf(df_train):
  category_club =
df_train.groupby('Category')['Text'].agg(sum).reset_index()
  postings = posting_list(category_club)
  doc_postings = doc_posting_list(df_train)
  tf_idf_matrix = build_tf_idf_mat(postings, category_club, doc_postings,
df_train.shape[0])
  prob_cat = get_class_prob(df_train)
  return tf_idf_matrix, prob_cat


tf_idf_matrix, prob_cat = train_tf_idf(df_train)
y_predicted = predict(X_test, tf_idf_matrix, prob_cat)

print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```



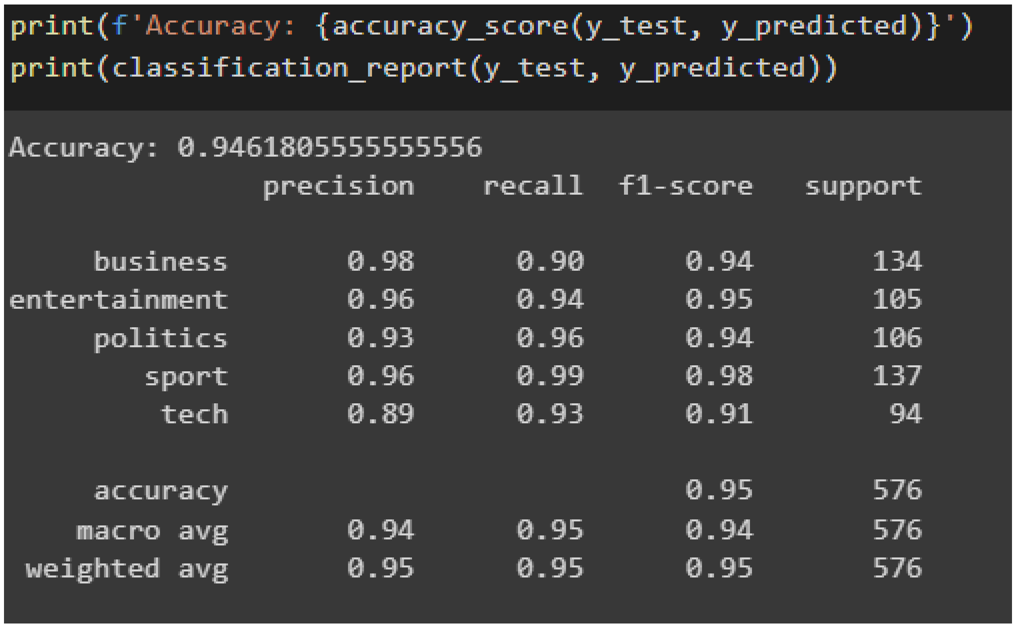
```
Accuracy: 0.9398148148148148
               precision    recall  f1-score   support

     business       0.98      0.94      0.96       100
entertainment       1.00      0.80      0.89        79
     politics       0.89      0.99      0.93        80
        sport       0.96      1.00      0.98       103
         tech       0.87      0.96      0.91        70

     accuracy                           0.94       432
    macro avg       0.94      0.94      0.93       432
 weighted avg       0.94      0.94      0.94       432
```

Inference

From the above results, we observed that tf-icf performed better than tf-idf because tf-icf Captures category-specific importance. That is, instead of across all documents as in TF-IDF, TF-ICF weights are dependent on how essential they are for a certain category. As a result, terms that are significant in one category but not another will be given more weight, which might increase the classification's accuracy. In multi-label classification problems, when a document may fall under more than one category, TF-ICF is a better fit. This is due to the fact that it can assign various weights to the same term for several categories, which can aid in capturing the nuances of the different labels.

Trying different split ratios to evaluate our model:

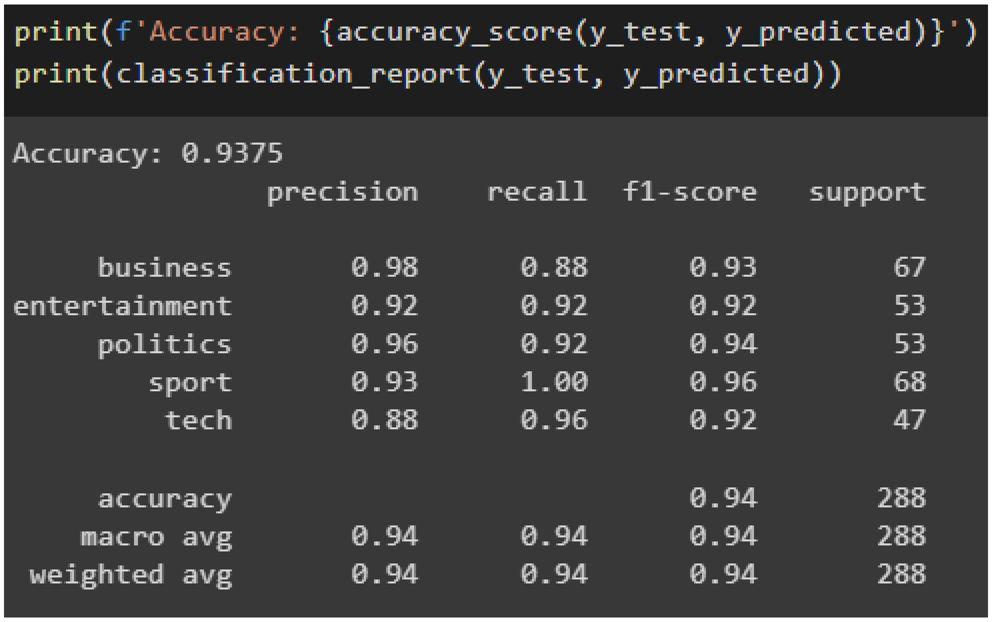
a. Performance when split ratio is 60:40:



```
print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

```
Accuracy: 0.9461805555555556
               precision    recall  f1-score   support

     business       0.98      0.90      0.94       134
entertainment       0.96      0.94      0.95       105
     politics       0.93      0.96      0.94       106
        sport       0.96      0.99      0.98       137
         tech       0.89      0.93      0.91        94

     accuracy                           0.95       576
    macro avg       0.94      0.95      0.94       576
 weighted avg       0.95      0.95      0.95       576
```

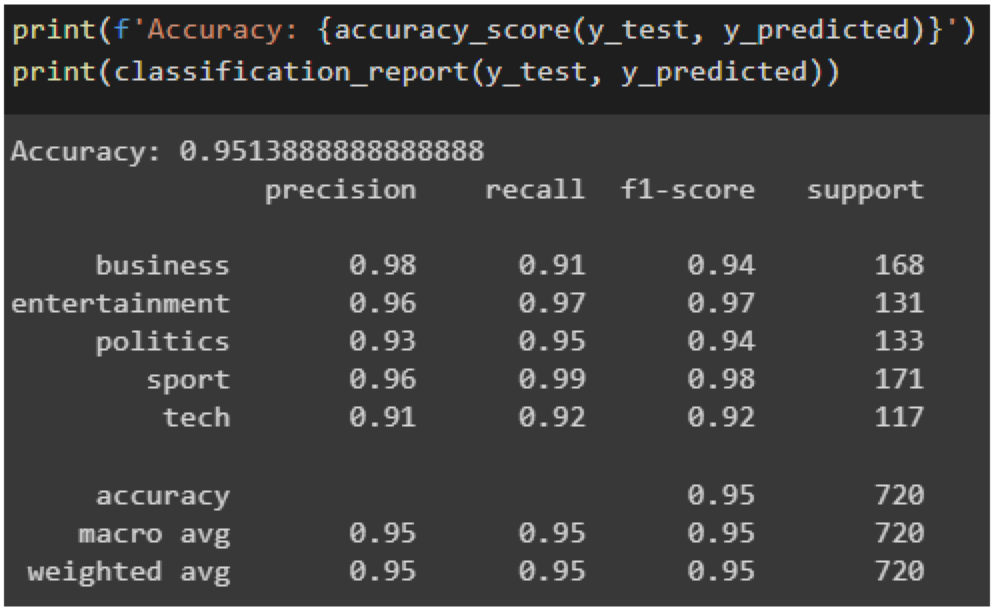
b. Performance when split ratio is 80:20:



```
print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

```
Accuracy: 0.9375
               precision    recall  f1-score   support

     business       0.98      0.88      0.93        67
entertainment       0.92      0.92      0.92        53
     politics       0.96      0.92      0.94        53
        sport       0.93      1.00      0.96        68
         tech       0.88      0.96      0.92        47

     accuracy                           0.94       288
    macro avg       0.94      0.94      0.94       288
 weighted avg       0.94      0.94      0.94       288
```

c. Performance when split ratio is 50:50:



```
print(f'Accuracy: {accuracy_score(y_test, y_predicted)}')
print(classification_report(y_test, y_predicted))
```

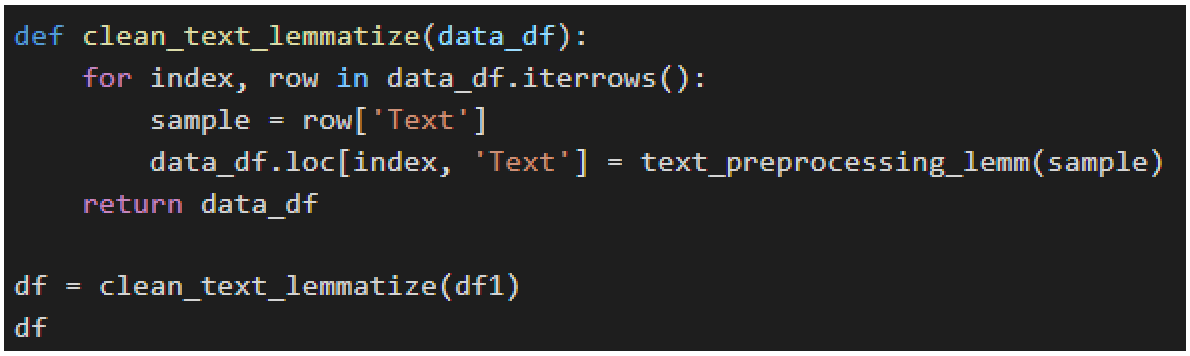
```
Accuracy: 0.9513888888888888
               precision    recall  f1-score   support

     business       0.98      0.91      0.94       168
entertainment       0.96      0.97      0.97       131
     politics       0.93      0.95      0.94       133
        sport       0.96      0.99      0.98       171
         tech       0.91      0.92      0.92       117

     accuracy                           0.95       720
    macro avg       0.95      0.95      0.95       720
 weighted avg       0.95      0.95      0.95       720
```

Inference

After analysing, we discovered that the accuracy increases with the amount of data, or the training data, increases since less overfitting would occur because there is a wider variety of data available to train. The different data then improves the test's accuracy. Similar to this, less training data is typically not as diverse, leading to inferior output performance. In addition, stratifying the dataset based on classes is a good approach to train the model because the distribution of the training and testing data should be comparable.

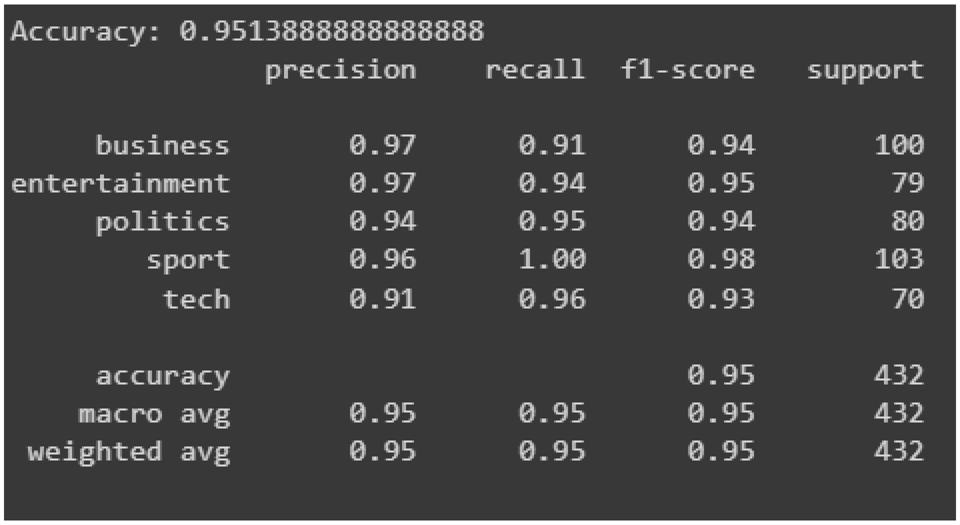
We also tried different preprocessing techniques to check and improve the performance of our classifier.

We tried lemmatization instead of stemming.



```
def clean_text_lemmatize(data_df):
    for index, row in data_df.iterrows():
        sample = row['Text']
        data_df.loc[index, 'Text'] = text_preprocessing_lemm(sample)
    return data_df

df = clean_text_lemmatize(df1)
df
```

Performance of the model when the split ratio was 70:30:



```
Accuracy: 0.9513888888888888
               precision    recall  f1-score   support

     business       0.97      0.91      0.94       100
entertainment       0.97      0.94      0.95        79
     politics       0.94      0.95      0.94        80
        sport       0.96      1.00      0.98       103
         tech       0.91      0.96      0.93        70

     accuracy                           0.95       432
    macro avg       0.95      0.95      0.95       432
 weighted avg       0.95      0.95      0.95       432
```

Inference

We observed that using stemming leads to better results as compared to lemmatization. Reason behind the same may be that by catching variations of the same word, such as plurals or verb tenses, stemming increases the recall (the percentage of relevant documents retrieved).

3.

About Dataset

In this question we had to use the Microsoft Learning to Rank dataset. The dataset contains various queries and their associated URLs with relevance judgement labels as relevance scores. We focused only on the queries with qid:4 and used the relevance judgement labels as the relevance score.

The query-url pair file is made using the order of max DCG.

```
def get_count_files(pair_list):
    rel_0 = 0
    rel_1 = 0
    rel_2 = 0
    rel_3 = 0
    rel_4 = 0
    for i in pair_list:
        if i[0] == '0':
            rel_0 = rel_0 + 1
        elif i[0] == '1':
            rel_1 = rel_1 + 1
        elif i[0] == '2':
            rel_2 = rel_2 + 1
        elif i[0] == '3':
            rel_3 = rel_3 + 1
        elif i[0] == '4':
            rel_4 = rel_4 + 1
    return
math.factorial(rel_0)*math.factorial(rel_1)*math.factorial(rel_2)*math.fac
torial(rel_3)*math.factorial(rel_4)

def arrange_max_dcg(pair_list):
    sorted_list = sorted(pair_list, key=lambda x:(x[0], x[1]),
reverse=True)
    file = open("output.txt","w")
    for i in sorted_list:
        file.write(i[1])
    file.close()
    return get_count_files(pair_list)

print("Number of files that can be created are: ")
print(arrange_max_dcg(url_list))
```

The number of such files which can be created are:

```
19893497375938370599826047614905329896936840170566570588205180312704857992
695193482412686565431050240000000000000000000000
```

Formula for DCG:

$$DCG = \sum_{i=1}^{5} \frac{rel_i}{\log_2(i+1)}$$

Our alternate formula for calculating DCG:

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log(1+i)}$$

The nDCG and DCG are calculated as follows(using both the formulae):

```python
def dcg_1(relevance_dict):
    ans = 0
    for i in range(1,len(relevance_dict)+1):
        ans = ans + (float(relevance_dict[i])/math.log2(i+1))
    return ans

def dcg_2(relevance_dict):
    ans = 0
    for i in range(1,len(relevance_dict)+1):
        ans = ans + ((pow(2,float(relevance_dict[i]))-1)/math.log2(i+1))
    return ans

def ndcg_1(relevance_dict, sorted_dict):
    dcg = dcg_1(relevance_dict)
    idcg = dcg_1(sorted_dict)
    return dcg/idcg

def ndcg_2(relevance_dict, sorted_dict):
    dcg = dcg_2(relevance_dict)
    idcg = dcg_2(sorted_dict)
    return dcg/idcg

sorted_dict = {}
sorted_list = sorted(rel_dict.items(), key=lambda x:x[1], reverse=True)
for i in range(len(sorted_list)):
    sorted_dict[i+1] = sorted_list[i][1]
print("nDCG")
print("For entire document: ", ndcg_1(rel_dict, sorted_dict))

sorted_dict_50 = {}
rel_dict_50 = {}
for i in range(50):
    sorted_dict_50[i+1] = sorted_list[i][1]
for i in range(1,51):
    rel_dict_50[i] = rel_dict[i]
print("At position 50: ", ndcg_1(rel_dict_50, sorted_dict_50))

print("\nnDCG using another formula")
print("For entire document: ", ndcg_2(rel_dict, sorted_dict))
print("At position 50: ", ndcg_2(rel_dict_50, sorted_dict_50))
```

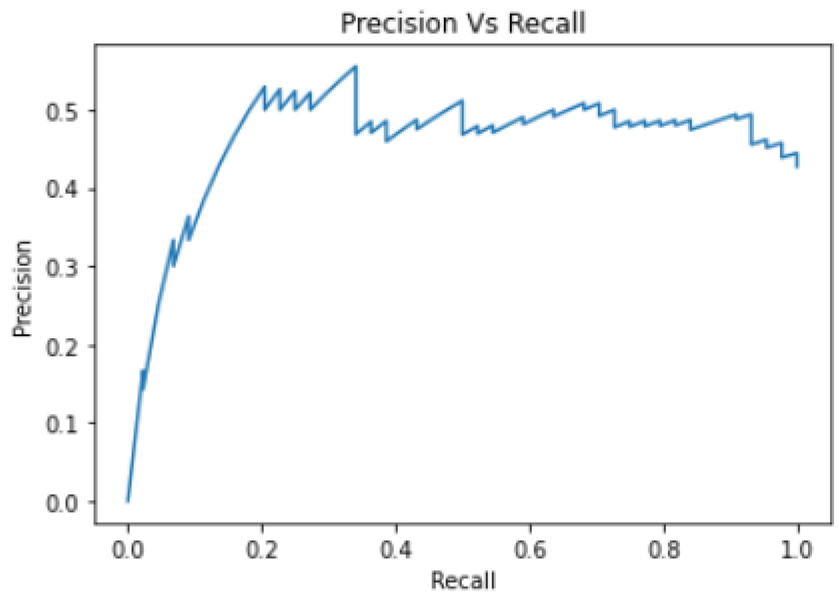
The result of the above code is:

```
nDCG
For entire document:  0.6357153091990775
At position 50:  0.37071213897397365

nDCG using another formula
For entire document:  0.5784691984582591
At position 50:  0.35612494416255847
```

The precision and recall are calculated as follows:

```python
precision = []
recall = []
cum = 0
counter = 0

sorted_rel_75 = sorted(rel_75, key=lambda x:x[1], reverse=True)
for tup in sorted_rel_75:
    counter= counter+1
    recall.append(cum/total_rel)
    precision.append(cum/counter)
    if tup[0]>'0':
        cum = cum+1

plt.plot(recall, precision)
plt.title('Precision Vs Recall')
plt.ylabel('Precision')
plt.xlabel('Recall')
plt.show()
```

The output graph is:



Here, we have taken a model that simply ranks URLs on the basis of the value of feature 75 (sum of TF-IDF on the whole document). So here the higher the value of the feature 75, the more relevant the URL.