Name: Devasy Patel

Roll No: 20BCE057

Practical 5: Consider a corpus of N documents. Implement Vector Space model (TFIDF consider normalized term frequency). Your implemented vector space model should rank the relevant retrieved documents by processing query.

In [ ]:
```
# tf-idf
# ranked retrieval model

import nltk
import numpy as np
import pandas as pd
# https://www.kaggle.com/edchen/tf-idf



# https://www.kaggle.com/divsinha/sentiment-analysis-countvectorizer-tf-idf



# https://www.kaggle.com/yassinehamdaoui1/creating-tf-idf-model-from-scratch



# https://www.kaggle.com/paulrohan2020/tf-idf-tutorial

# https://www.kaggle.com/adamschroeder/countvectorizer-tfidfvectorizer-predict-comments
```

In [ ]:
```
corpus = [
    "Hello there, how are you?","My name is Amithab and I am a data scientist","I am a data scientist and I love to code in python","I love to code in python and I am a data scientist","I'm Ed Sheeran the famous singer and I love to code in python"
    ]
```

In [ ]:
```
class TfidfVecorizer:
    def fit(self, X):
        X = [doc.lower() for doc in X]
        # make colums as unique words
        unique_words = set([word for doc in X for word in doc.split()])
        # make rows as documents
        docs = [doc for doc in X]
        # compute idf for each word
        self.word2tfidf = {word: self.compute_idf(word, X) for word in unique_words}

        # comute tfidf for each word in each document
        self.doc2tfidf = {i: {word: self.compute_tfidf(word, doc) for word in doc} for i, doc in enumerate(docs)}

        return self.doc2tfidf


    def compute_tfidf(self, word, doc):
        tf = doc.split().count(word) / len(doc.split())
        idf = self.compute_idf(word, doc)
        return tf * idf



    def compute_idf(self, word, X):
        return np.log(len(X) / (1 + sum([word in doc.split() for doc in X])))
    def transform(self, X):
        X = [doc.lower() for doc in X]
        return [[self.compute_tfidf(word, doc) for word in doc.split()] for doc in X]
```

In [ ]:
```
# convert to lower case
corpus = [doc.lower() for doc in corpus]

# make tf-idf matrix without using library
# https://www.kaggle.com/paulrohan2020/tf-idf-tutorial

query = "I love to code in R and I am a data scientist"

tfidf = TfidfVecorizer()
response = tfidf.fit(corpus)
print(response)
response2 = tfidf.transform(corpus)
response2 = pd.DataFrame(response2)
response2
```

```
{0: {'h': 0.0, 'e': 0.0, 'l': 0.0, 'o': 0.0, ' ': 0.0, 't': 0.0, 'r': 0.0, ',': 0.0, 'w': 0.0, 'a': 0.0, 'y': 0.0, 'u': 0.0, '?': 0.0}, 1: {'m': 0.0, 'y': 0.0, ' ': 0.0, 'n': 0.0, 'a': 0.1586965056582042, 'e': 0.0, 'i': 0.19924301646902062, 's': 0.0, 't': 0.0, 'h': 0.0, 'b': 0.0, 'd': 0.0, 'c': 0.0}, 2: {'i': 0.3533772503334846, ' ': 0.0, 'a': 0.17668862801667423, 'm': 0.0, 'd': 0.0, 't': 0.0, 's': 0.0, 'c': 0.0, 'e': 0.0, 'n': 0.0, 'l': 0.0, 'o': 0.0, 'v': 0.0, 'p': 0.0, 'y': 0.0, 'h': 0.0}, 3: {'i': 0.3533772560334846, ' ': 0.0, 'l': 0.0, 'o': 0.0, 'v': 0.0, 'e': 0.0, 't': 0.0, 'c': 0.0, 'd': 0.0, 'n': 0.0, 'p': 0.0, 'y': 0.0, 'h': 0.0, 'a': 0.17668862801667423, 'm': 0.0, 's': 0.0}, 4: {'i': 0.19241815013378546, "'": 0.0, 'm': 0.0, ' ': 0.0, 'e': 0.0, 'd': 0.0, 's': 0.0, 'h': 0.0, 'r': 0.0, 'a': 0.0, 'n': 0.0, 't': 0.0, 'f': 0.0, 'o': 0.0, 'u': 0.0, 'g': 0.0, 'l': 0.0, 'v': 0.0, 'c': 0.0, 'p': 0.0, 'y': 0.0}}
```

Out[ ]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.643775	0.643775	0.643775	0.643775	0.643775	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	0.378419	0.378419	0.378419	0.378419	0.378419	0.199243	0.378419	0.158697	0.378419	0.378419	NaN	NaN	NaN
2	0.353377	0.326002	0.176689	0.326002	0.326002	0.326002	0.353377	0.326002	0.326002	0.326002	0.326002	0.326002	NaN
3	0.353377	0.326002	0.326002	0.326002	0.326002	0.326002	0.326002	0.353377	0.326002	0.176689	0.326002	0.326002	NaN
4	0.316221	0.316221	0.316221	0.316221	0.316221	0.316221	0.316221	0.192418	0.316221	0.316221	0.316221	0.316221	0.316221

In [ ]:
```
response2.fillna(0, inplace=True)
```

In [ ]:
```
# add query to the corpus
corpus.append(query)
response2 = tfidf.transform(corpus)
response2 = pd.DataFrame(response2)
response2.fillna(0, inplace=True)
response2
```

Out[ ]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.643775	0.643775	0.643775	0.643775	0.643775	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1	0.378419	0.378419	0.378419	0.378419	0.378419	0.199243	0.378419	0.158697	0.378419	0.378419	0.000000	0.000000	0.000000
2	0.353377	0.326002	0.176689	0.326002	0.326002	0.326002	0.353377	0.326002	0.326002	0.326002	0.326002	0.326002	0.000000
3	0.353377	0.326002	0.326002	0.326002	0.326002	0.326002	0.326002	0.353377	0.326002	0.176689	0.326002	0.326002	0.000000
4	0.316221	0.316221	0.316221	0.316221	0.316221	0.316221	0.316221	0.192418	0.316221	0.316221	0.316221	0.316221	0.316221
5	0.335817	0.317222	0.317222	0.317222	0.317222	0.259460	0.317222	0.335817	0.317222	0.167909	0.317222	0.317222	0.000000

In [ ]:
```
# now for ranked retrieval model of the query
#  measure the similarity between the query and each document in the corpus using cosine similarity
def cosine_similarity(vector1, vector2):
    dot_product = np.dot(vector1, vector2)
    norm_vector1 = np.linalg.norm(vector1)
    norm_vector2 = np.linalg.norm(vector2)
    return dot_product / (norm_vector1 * norm_vector2)

# compute cosine similarity between query and each document in the corpus
cosine_similarities = []
for i in range(len(corpus)):
    cosine_similarities.append(cosine_similarity(response2.iloc[-1], response2.iloc[i]))

# get the index of the most similar document except the last one which is the query
most_similar_doc_index = np.argmax(cosine_similarities[:-1])
print("The most similar document to the query is document number: ", most_similar_doc_index)
```

```
The most similar document to the query is document number:  3
```