Information Retrieval (IT458)

Assignment 1: Term weighting and VSM

Submitted by: Jaidev Chittoria
Roll No.: 181IT119

Date: 10th October

Corpus:

For this assignment we used a corpus of around 690 documents related to physics lectures.

Preprocessing Techniques:

The Preprocessing Techniques which were applied to the corpus are as follows

- Removal of Punctuation signs like (!()-[]{};:'"\ etc.
- Converting the text to lowercase.
- Removing Stop words (The list of stopwords is extracted using the nltk package of Python).
- Stemming : All the morphological variants of the word are reduced to the root word in the document. Stemming helps to reduce multiple morphological variants into a common word which in turn helps to reduce the size of inverted index.

In [269]:
```
def tokenize(data):
    lines = data.lower()
    lines = re.sub('[^A-Za-z]+', ' ', lines)
    tokens = lines.split()
    clean_tokens = [word for word in tokens if word not in stop_list]
    stem_tokens = [st.stem(word) for word in clean_tokens]
    clean_stem_tokens = [word for word in stem_tokens if word not in stop_list]
    clean_stem_tokens = ' '.join(map(str,  clean_stem_tokens))
    clean_stem_tokens = shortword.sub('', clean_stem_tokens)
    return clean_stem_tokens

def extractTokens(beautSoup, tag):
    textData = beautSoup.findAll(tag)
    textData = ''.join(map(str, textData))
    textData = textData.replace(tag, '')
    textData = tokenize(textData)
    return textData
```

In [270]:
```
for fname in filenames:
    infilepath = in_path + '/' + fname
    outfilepath = out_path + '/' + fname
    with open(infilepath) as infile:
        with open(outfilepath, 'w') as outfile:
            fileData = infile.read()
            soup = BeautifulSoup(fileData)
            title = extractTokens(soup, 'title')
            text = extractTokens(soup, 'text')
            outfile.write(title)
            outfile.write(" ")
            outfile.write(text)
        outfile.close()
    infile.close()
```

Term weight Modeling:

It is used for evaluating the usefulness of a term in a particular document. A term which is occurring in very few documents is important in determining the document.

- After the preprocessing of the corpus and the query, we will convert the documents into vocabulary.
- We will calculate the DF value (Document Frequency) for each term in the vocabulary.
- Finally using the generated DF values we will calculate the TF-IDF value of each term in the vocabulary.
- TF-IDF=TF x IDF
- There are various schemes for calculating TF and IDF for now we will be using the standard scheme.

In [278]:
```
tf_idf = {}

def log_inverse_tfIdf():
    doc = 0
    for i in range(no_of_docs):
        tokens = all_docs[i].split()
        counter = Counter(tokens)
        words_count = len(tokens)

        for token in np.unique(tokens):
            tf = counter[token]/words_count
            df = DF[token] if token in vocab else 0
            idf = np.log((no_of_docs)/(df))
            tf_idf[doc, token] = tf*idf
```

Snap of output :

```
In [279]: tf_idf
Out[279]: {(0, 'air'): 0.0050107854092344365,
           (0, 'also'): 0.0062631961755327745,
           (0, 'applic'): 0.0053163529797158795,
           (0, 'atmospher'): 0.006567027891899958,
           (0, 'axisymmetr'): 0.0028088656526642874,
           (0, 'basi'): 0.0031945176774437487,
           (0, 'blunt'): 0.012906227010589333,
           (0, 'bodi'): 0.02659893421343498,
           (0, 'case'): 0.014157828590568753,
           (0, 'composit'): 0.0023752705220474496,
           (0, 'cone'): 0.003648869637149201,
           (0, 'consid'): 0.00549633367805113,
           (0, 'constant'): 0.009869612980311427,
           (0, 'densiti'): 0.0037029219701248 4,
           (0, 'edg'): 0.00488421314279944,
           (0, 'equat'): 0.007618022103224242,
           (0, 'equilibrium'): 0.0063689035354887497,
           (0, 'examin'): 0.006017528092524547,
           (0, 'exist'): 0.00388898231478357,
```

Represent Document corpus using the standard TF-IDF weights considering Vector space model

Document vector

Forming document vectors using the tf-idf values

```
In [31]: D = np.zeros((no_of_docs, vocab_size))
         for i in tf_idf:
             ind = vocab.index(i[1])
             D[i[0]][ind] = tf_idf[i]
```

```
In [32]: import numpy as np

         print(np.matrix(D))

[[0.16819812 0.07068877 0.0355104  ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
```

Used queries

```
In [36]: query_file = open(preproc_query, 'r')
         queries = query_file.readlines()
         queries
Out[36]: ['investig made wave system creat static pressur \n',
          'vortic heat transfer\n',
          'absenc vortic\n',
          'gener effect flow field\n',
          '\n']
```

Relevance rankings of documents in a keyword search can be calculated, using the assumptions of document similarity theory, by comparing the deviation of angles between each document vector and the original query vector where the query is represented as a vector with same dimension as the vectors that represent the other documents.

So we will use cosine similarity as a closeness measure for generating the rank list for given queries.

Cosine similarity is the cosine of the angle between two *n*-dimensional vectors in an *n*-dimensional space. It is the dot product of the two vectors divided by the product of the two vectors' lengths (or magnitudes).
First we will calculate it between two vectors and follow the same procedure for calculating similarity between document vector and query vector.
Below is a snap of the process

In [34]:
```
def cosine_sim(x, y):
    cos_sim = np.dot(x, y)/(np.linalg.norm(x)*np.linalg.norm(y))

    return cos_sim
```

In [35]:
```
def cosine_similarity(k, query):
    tokens = query.split()
    d_cosines = []
    query_vector = gen_vector(tokens)

    for d in D:
        d_cosines.append(cosine_sim(query_vector, d))

    if k == 0:
        out = np.array(d_cosines).argsort()[::-1]
    else:
        out = np.array(d_cosines).argsort()[-k:][::-1]
    return out
```

For first query

```
[[0,
  array([461, 468, 164, 307, 372, 248,  37, 341, 237, 176, 446, 194, 137,
         331,   1, 285,  90, 236, 549, 638, 147, 348, 553, 181, 502, 643,
         519, 379, 208, 352, 122, 197, 598, 337, 126, 421, 543, 548, 242,
         617, 264, 470, 531, 193, 625,   9, 310, 357, 494, 185,  34, 241,
         389, 162, 597, 659,   5, 469, 476, 283, 589, 411, 277, 426, 608,
          79, 261, 355, 161, 329, 550, 165,  11, 204,   8, 128, 201, 425,
         490, 117, 251, 414, 347, 537, 220, 282, 124, 576, 439, 246, 190,
         525,  98,  96, 376, 163, 437, 325, 308, 272,  24, 395, 660, 507,
         672,  71, 247, 547, 330,  25, 344, 301,  86,  87, 150, 641,   4,
         647, 110, 610, 654, 378, 104, 358, 271,  76, 311, 323, 653, 166,
```

For second query

```
[1,
 array([ 84, 371, 108, 435, 675, 229, 464, 462, 478,  12,  32, 128, 554,
        540, 341, 506, 635, 573, 655, 532, 369, 427, 648, 684, 143, 187,
        564, 516, 233, 130, 100, 372, 538,  40, 267, 182, 106,  14, 171,
         22,   7, 344, 588, 599, 562, 192,  71, 398, 613,  72, 500, 473,
        305, 184, 418, 432,  54, 545, 456, 375, 428, 671, 298, 658, 284,
        544, 489,  39, 513, 397, 529, 677, 689, 518, 392, 231, 329, 145,
        230, 302, 601, 501, 222, 414, 488, 396, 175, 363, 404, 235, 583,
        531,  67, 362, 586, 205, 451, 335, 239,  59, 453, 315, 496, 580,
        472, 103,  77, 248,  95, 336, 199, 361, 365,  28, 637, 110, 480,
```

For third query

```
 [2,
  array([ 84, 108, 371, 435,  27, 140, 478,  32, 341, 573, 655, 684, 372,
         171, 344, 192,  71,  72,  54, 544, 489,  39, 298, 506, 145, 414,
         363, 362, 586, 451, 361, 385, 147, 653, 583, 560, 309,  86,  95,
         229, 225, 226, 224, 228, 223, 227, 236, 230, 240, 247, 246, 245,
         244, 243, 242, 241, 239, 231, 238, 237, 221, 235, 234, 233, 232,
         222, 212, 220, 189, 197, 196, 195, 194, 193, 191, 190, 188, 199,
         187, 186, 185, 184, 183, 182, 181, 198, 200, 219, 210, 218, 217,
         216, 215, 214, 213, 211, 209, 201, 208, 207, 206, 205, 204, 203,
```

Experiment with different TF-IDF schemes

Below are three different schemes combination is used

```
def log_inverse_tfIdf():
    doc = 0
    for i in range(no_of_docs):
        tokens = all_docs[i].split()
        counter = Counter(tokens)
        words_count = len(tokens)

        for token in np.unique(tokens):
            tf = counter[token]/words_count
            df = DF[token] if token in vocab else 0
            idf = np.log((no_of_docs)/(df))
            tf_idf[doc, token] = tf*idf

        doc += 1

def log_inverse_smooth_tfIdf():
    doc = 0
    for i in range(no_of_docs):
        tokens = all_docs[i].split()
        counter = Counter(tokens)
        words_count = len(tokens)

        for token in np.unique(tokens):
            tf = counter[token]/words_count
            df = DF[token] if token in vocab else 0
            idf = np.log(1+(no_of_docs)/(df))
            tf_idf[doc, token] = tf*idf

        doc += 1

def log_prob_tfIdf():
    doc = 0
    for i in range(no_of_docs):
        tokens = all_docs[i].split()
        counter = Counter(tokens)
        words_count = len(tokens)

        for token in np.unique(tokens):
            tf = counter[token]/words_count
            df = DF[token] if token in vocab else 0
            idf = np.log((no_of_docs-df)/(df))
            tf_idf[doc, token] = tf*idf
```

Combinations

First- Binary + inverse frequency
Second- Binary + inverse Frequency Smooth
Third- Binary + probabilistic inv Frequency (all in order)

First is same as above

Second combination:

First query

```
[[0,
  array([468, 461, 164, 307, 372, 248,  37, 341, 237, 446, 176, 137, 194,
         331,   1,  90, 285, 638, 236, 147, 549, 643, 181, 348, 502, 379,
         122, 553, 208, 519, 421, 352, 598, 197, 337, 126, 242, 543, 264,
         470, 617, 548, 193, 310, 531, 357, 625, 494, 241,   9, 589, 476,
         162,  34, 469, 185, 389,   5, 659, 426, 283, 608, 597, 411, 277,
         537,  11, 261,   8, 550, 201, 355, 490,  79, 439, 161, 165, 251,
         128, 576, 329, 282, 204, 220, 425, 190, 347, 414, 507, 117, 247,
         325, 124, 376,  96, 308, 150,  98, 163, 395,  71, 272, 437, 654,
         525, 547, 246, 641, 660, 672,  25, 330, 110,  24, 430, 647,  87,
         104, 344,  76, 114, 301,   4,  48,  86, 378, 271, 323, 166, 610,
         358, 173, 653, 311, 685, 223, 514,  60,   0, 157, 399, 491, 590,
```

Second Query

```
[1,
 array([ 84, 371, 108, 435, 675, 229, 464, 462,  12, 554, 128, 540,  32,
        635, 478, 532, 506, 341, 427, 648, 573, 369, 655, 130, 143, 233,
        538, 187, 516, 564, 100,  40,  22, 267, 182, 588, 106,   7,  14,
        599, 562, 372, 684, 613, 500, 398, 171, 344, 184, 473, 305, 192,
        432, 671, 456, 658, 428,  71, 284, 418, 375,  72, 545,  54, 513,
        518, 677, 529, 397, 298, 329, 392, 231, 689, 230, 544, 601, 489,
        302,  39, 488, 222, 501, 396, 145, 175, 235, 404, 414, 531, 205,
         67, 335, 239, 583, 363, 496,  59, 315, 362, 453, 451, 472, 103,
        580, 586,  77, 248,  95, 336, 199, 365, 480,  28, 353, 110, 440,
        637, 210, 510,  49, 391, 159, 627, 320, 361, 280,  31, 610, 619,
```

Third Combination:

First query

```
[[0,
  array([461, 468, 164, 307, 372, 248,  37, 341, 237, 194, 176, 446, 236,
         285, 549,   1, 331, 553,  90, 638, 348, 147, 181, 502, 519, 137,
         352, 208, 197, 643, 548, 122, 598, 337, 379, 126, 543, 617, 185,
           9, 470, 531, 264, 242, 421, 625, 494,  34, 597, 389, 357, 659,
         193,   5, 283, 411, 162, 310, 469,  79, 329, 476, 161, 204, 355,
         261, 425, 165, 117, 241, 414, 246, 347, 550, 277, 589, 128, 525,
         124, 251,  11,  24, 201, 220,  98, 437, 282, 490,  96,   8, 163,
         608, 376, 426, 301,  86, 190, 308, 272, 660, 344, 330, 610, 576,
         672, 325,  25,  87,   4, 358, 439, 395, 537,   0, 378, 311, 271,
```

Second Query

```
 [1,
  array([ 84, 108, 371, 435, 675, 478,  32, 229, 341, 464, 684, 655, 462,
         573,  12, 128, 506, 554, 540, 635, 372, 171, 532, 369, 344, 648,
         427, 564, 143, 187, 100, 516,  71, 192, 233, 267,  40,  14, 130,
         106,  72, 182,   7, 538,  54,  22, 588, 599, 298, 562, 544, 398,
         489, 545, 418, 375,  39, 432, 305, 613, 473, 456, 428, 145, 500,
         671, 184, 284, 658, 689, 363, 397, 414, 513, 529, 677, 392, 231,
         518, 302, 329, 362, 230, 601, 586, 501, 583, 404, 222, 396, 175,
```

As we can see there is a slight change in the rankings obtained with different TF-IDF Schemes