Text feature extraction TF-IDF

In information retrieval the *term frequency – inverse document frequency* (also called tf-idf) is a well know method to evaluate how important is a word in a document.

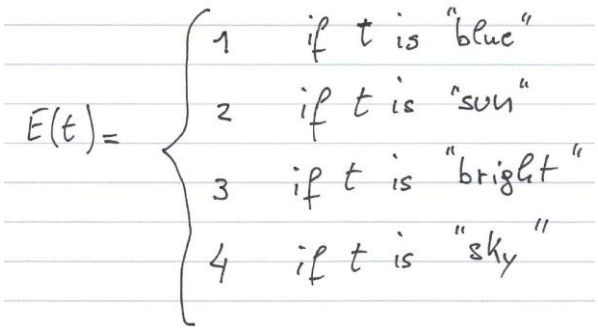
Going to the vector space

The first step in modeling the document into a vector space is to create a dictionary of terms present in documents. To do that you need to extract all terms from the document and convert them to the dimensions in the vector space. We know that there are words (called stop words) like “the”, “is”, “at”, “on” that are present in almost all documents. Since our aim is to extract important features from documents, using stop words does not help us in the information extraction; so we will ignore them.

Let’s take the four documents below to define our collection of documents.

```
Document collection
d1: The sky is blue.
d2: The sun is bright.
d3: The sun in the sky is bright.
d4: We can see the shining sun, the bright sun.
```

What we have to do is to create a index vocabulary (dictionary) of the words occurring in the documents in the collection Z. We will use the following index vocabulary denoted as E(t) where the t is the term:



Note that the terms like “is” and “the” were ignored since they are stop words. Now that we have an index vocabulary, we can convert the test document set into a vector space where each term is indexed as our index vocabulary. The first element of the vector represents the term “blue”, the second represents “sun” and so on. Next we use the term-frequency weight to represent each term in our vector space. Term-frequency is nothing more than a measure of how many times the terms present in our vocabulary E(t) are present in each document. The term-frequency is defined as a couting function:

$$tf(t,d) = \sum_{x \in d} fr(x,t)$$

where the fr(x, t) is a function defined as:

$$fr(x,t) = \begin{cases} 1 & \text{if } x=t \\ 0 & \text{otherwise} \end{cases}$$

So, what the tf(t,d) returns is how many times the term t is present in the document d. For example, tf("sun", d4) = 2 since we have two occurrences of the term “sun” in the document d4.

Now we can go on into the creation of the document vector which is represented by:

$$\vec{v_d} = \left(tf(t_1,d), tf(t_2,d), \ldots, tf(t_n,d) \right)$$

Each dimension of the vector represents a term of the vocabulary; for example tf(t1,d2) that is the first dimension of the vector of d2, represents the frequency of the term “blue”.

The vector representation of our document collection is:

$$\vec{v}_{d_1} = (0,0,0,1)$$

$$\vec{v}_{d_2} = (0,1,1,0)$$

$$\vec{v}_{d_3} = (0,1,1,1)$$

$$\vec{v}_{d_4} = (0,2,1,0)$$

Since we have a collection of documents, we can represent them as a matrix |Z| x F, where |Z| is the cardinality of the document space (how many documents we have) and the F is vocabulary size.

The matrix representation of Z is:

$$M_{|Z| \times F} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$$

As you may have noted, these matrices representing the term frequencies tend to be very **sparse** with majority of terms being zeros, and that's why you'll see a common representation of these matrix as sparse matrices.

Python implementation

The vector space conversion can easily be implemented in Python by using the scikit.learn module. We start by defininining the document collection Z.

```
d1 = "The sky is blue."
d2 = "The sun is bright."
d3 = "The sun in the sky is bright."
d4 = "We can see the shining sun, the bright sun."
Z = (d1,d2,d3,d4)
```

In scikit.learn, what we have presented as the term-frequency, is called **CountVectorizer**, so we need to import it and create a new instance:

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
```

The CountVectorizer already performs to lowercase text conversion, accents removal, token extraction and filter stop words. You can see more information by printing the vectorizer information:

```
print(vectorizer)
```

Output:

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
      dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
      lowercase=True, max_df=1.0, max_features=None, min_df=1,
      ngram_range=(1, 1), preprocessor=None, stop_words=None,
      strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
      tokenizer=None, vocabulary=None)
```

As you can see in the output above, the tokenizer does not have any vocabulary and stop words (`stop_words=None,vocabulary=None`). Let's create our own vocabulary and stop words set.

```
my_stop_words={"the","is"}
my_vocabulary={'blue': 0, 'sun': 1, 'bright': 2, 'sky': 3}
vectorizer=CountVectorizer(stop_words=my_stop_words,vocabulary=my_vocabulary)
```

You can print the stop words list and the vocabulary as:

```
print(vectorizer.vocabulary)
print(vectorizer.stop_words)
```

Let's use our vectorizer now to create the sparse matrix of the document set:

```
smatrix = vectorizer.transform(Z)
print(smatrix)
```

Output:

```
  (0, 0)        1
  (0, 3)        1
  (1, 1)        1
  (1, 2)        1
  (2, 1)        1
  (2, 2)        1
  (2, 3)        1
  (3, 1)        2
  (3, 2)        1
```

Note that the sparse matrix created called smatrix is a Scipy sparse matrix with elements stored in a coordinate format. But you can convert it into a dense format:

```
matrix = smatrix.todense()
print(matrix)
```

Output:

```
  matrix([[1, 0, 0, 1],
          [0, 1, 1, 0],
          [0, 1, 1, 1],
          [0, 2, 1, 0]], dtype=int64)
```

Note that the matrix created has the format |Z| x F.

The use of the term frequency could lead us to problems like:

i. The keyword spamming where we have a term in a document that is repeated many times with the sole purpose of improving the document ranking on an IR (Information Retrieval) system.

ii. The creation of a bias towards long documents making them look more important of what they are simply because of the high frequency of the terms in the document.

To address (ii) the term frequency tf(t,d) is usually normalized.

Computing the tf-idf score

The main problem with the term-frequency approach is that it scales up frequent terms and scales down rare terms which are empirically more informative than the high frequency terms. The basic intuition is that a term that occurs frequently in many documents is not a good discriminator. The important question here is: why would you in a classification problem emphasize a term which is almost present in the entire corpus of your documents?

The tf-idf weight comes to solve these problems. What tf-idf gives is how important is a word to a document in a collection, and that's why tf-idf incorporates local and global parameters. It takes in consideration not only a term isolated to a document but also the term within the document collection. What tf-idf does to solve that problem is to scale down the frequent terms while scaling up the rare terms; a term that occurs 10 times more than another isn't 10 times more important than it, that's why tf-idf uses the logarithmic scale to do that.

The idf weight (inverse document frequency) is defined as:

$$idf(t) = \log \frac{|Z|}{1 + |\{d: t \in d\}|}$$

where |Z| is the total number of documents and |{d: t∈d}| is the number of documents where the term t occurs. We are adding 1 into the formula to avoid zero-division.

The formula for the tf-idf is:

$$tf\text{–}idf(t) = tf(t,d) * idf(t)$$

A high weight of the tf-idf value for a term t is reached when you have a high term frequency (tf) of t in the given document (local parameter) and a low document frequency of t in the whole collection (global parameter).

To calculate the tf-idf value of our documents we will use the TfidfTransformer module provided by Scikit.learn on the word counts (smatrix) we computed earlier.

```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer = TfidfTransformer(norm="l2")

tfidf_transformer.fit(smatrix)
```

Note that we have specified the norm as L2 to normalize; this is optional (actually the default is L2-norm). The fit() method calculates the idf values for the matrix smatrix and stors them in the internal attribute called idf_.

To get a glimse of how the idf values look, we are going to print them by placing the idf values in a python dataframe. The values will be sorted in ascending order.

```
# print idf values

feature_names = vectorizer.get_feature_names()

import pandas as pd

df_idf=pd.DataFrame(tfidf_transformer.idf_, index=feature_names,columns=["idf_weights"])

# sort ascending

df_idf.sort_values(by=['idf_weights'])
```

Output:

```
           idf_weights
  blue       1.916291
  sun        1.223144
  bright     1.223144
  sky        1.510826
```

Note that the words “sun” and “bright” have the lowest idf values. This is expected as these words appear in every document in our collection. The lowest the idf value of a word, the less unique the word is to any particular document.

Once you have the idf values, you can compute the tf-idf scores by invoking tfidf_transformer over the sparse matrix smatrix. This corresponds to compute the $tf * idf$ multiplication where the term frequency is weighted by its idf values.

```
# tf-idf scores

tf_idf_vector = tfidf_transformer.transform(smatrix)
```

Now we can print the tf-idf values of the first document "The sky is blue.". We place the tf-idf scores of the first document into a pandas dataframe and sort it in descending order.

```
# get tfidf vector for first document

first_document = tf_idf_vector[0]  # first document "The sky is blue."

# print the scores

df=pd.DataFrame(first_document.T.todense(), index=feature_names, columns=["tfidf"])

df.sort_values(by=["tfidf"],ascending=False)
```

Output:

```
           tfidf
blue    0.785288
sun     0.000000
bright  0.000000
sky     0.619130
```

Note that only certain words have scores. This is because our first document is “The sky is blue.” All the words in this document have a tf-idf score and everything else show up as zeros. The more common the word across documents, the lower its score and the more unique a word (e.g. “blu”) is to our first document the higher the score.

Document Similarity

Cosine Similarity for Vector Space Models

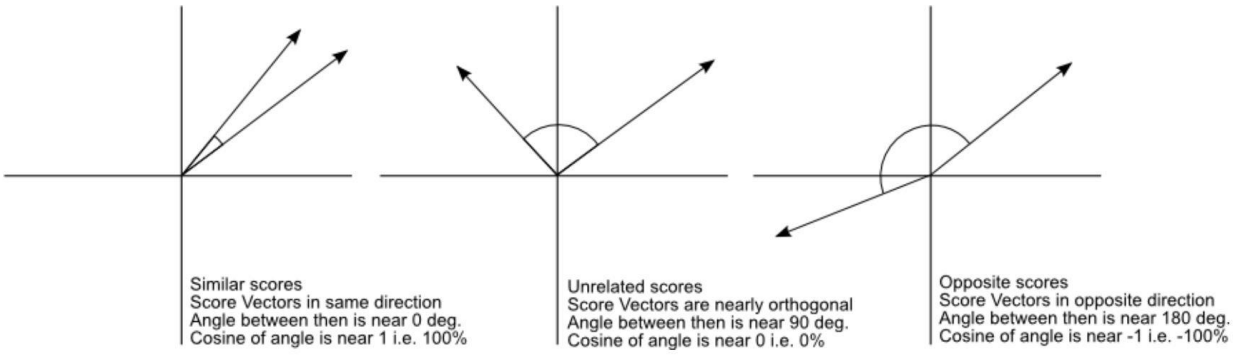
In the previous sections we learned how a document can be modeled in the Vector Space, how the TF-IDF transformation works and how the TF-IDF is calculated. Now we are going to learn how to use a well-known similarity measure (cosine similarity) to calculate the similarity between different documents.

The cosine similarity between two vectors (or two documents on the vector space) is a measure that calculates the cosine of the angle between them. This metric is a measurement of orientation and not magnitude, it can be seen as a comparison between documents on a normalized space because we're not taking into consideration the magnitude of each word count (tf-idf) of each document, but the angle θ between the documents. What we have to do to build the cosine similarity equation is to solve the equation of the dot product for the $\cos\theta$:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \left\|\vec{b}\right\| \cos\theta$$

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \left\|\vec{b}\right\|}$$

this is the cosine similarity formula. Cosine similarity will generate a metric that says how related two documents are by looking at the angle:

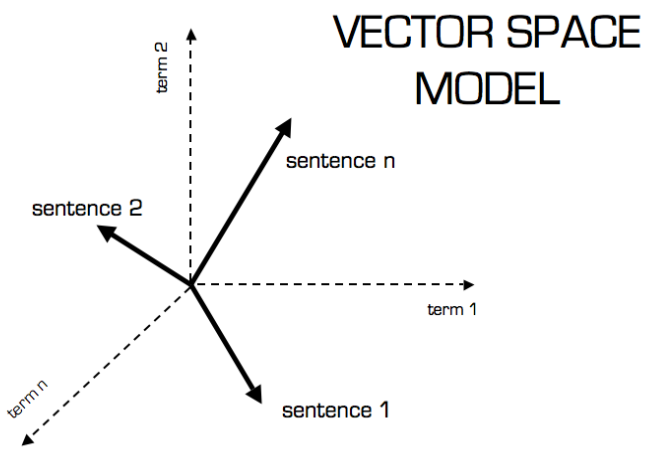


The cosine similarity values for different documents, 1 (same direction), 0 (90 deg.), -1 (opposite directions)

It is worth saying that even if we have a vector pointing to a point far from another vector, they still could have an small angle and that is the central point on the use of cosine similarity; the measurement ignores the higher term count on documents.

Suppose we have a document with the word “sky” appearing 200 times and another document with the word “sky” appearing 50, the Euclidean distance between them will be high but the angle will still be small because they are pointing to the same direction. This is what matters when we are comparing documents.

Now that we have a vector space model of documents (like on the image below) modeled as vectors (with TF-IDF counts) and also have a formula to calculate the similarity between documents in this space, let’s see how we can do it in practice using scikit-learn.



Let’s take the same documents as in the previous section.

```
d1 = "The sky is blue."
d2 = "The sun is bright."
d3 = "The sun in the sky is bright."
d4 = "We can see the shining sun, the bright sun."
Z = (d1,d2,d3,d4)
```

We need to transform each document into a count-vectorised form. To do so we will use the TF-IDF vectorizer. Here we use the fit_transform method because it fits the transformer with the supplied data and creates a TF-IDF matrix.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()

tfidf_matrix = tfidf_vectorizer.fit_transform(Z)

print(tfidf_matrix.shape)
```

Output:

```
(4, 11)
```

This means that we have created a TF-IDF matrix consisting of 4 rows (since we have 4 documents) and 11 columns (the number of tf-idf terms). For each document we calculate the cosine similarity between the selected document and the remaining documents. Let's select the first document "The sky is blue". That is, we take out the first row of the tfidf matrix (tfidf_matrix[0]) and calculate the cosine similarity with all the other rows.

```
from sklearn.metrics.pairwise import cosine_similarity

cos_similarity = cosine_similarity(tfidf_matrix[0], tfidf_matrix)

print(cos_similarity)
```

Output:

```
array([[1.        , 0.36651513, 0.52305744, 0.13448867]])
```

The first value of the array is 1.0 since it is the cosine similarity between the first document with itself. From the Output we can see that the third document "The sun in the sky is bright" is the most similar to the first document (cosine similarity=0.52). This result is expected since the first and third document contain similar words.

To calculate the angle θ you can solve the cosine similarity for the angle between the vectors $\vec{a}, \vec{b}$:

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \, \|\vec{b}\|}$$

$$\theta = arccos \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \, \|\vec{b}\|}$$

where the arccos is the same as the inverse of the cosine ($\cos^{-1}$).

Lets for instance calculate the angle θ between the first and third documents:

```
# Take the cos similarity of the third document (cos similarity=0.52)

angle_in_radians = math.acos(cos_similarity[2])

print(math.degrees(angle_in_radians))
```

Output:

```
58.4
```

This is the angle between the first and the third document of our document set.

Classifying Text

One machine learning method often used in text classification is multinomial naïve Bayes. We will use the sparse word count features from the 20 Newsgroups corpus to show how we can classify these documents.

Let's start by downloading the data set and take a look at the target names:

```
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups()

data.target_names
```

Output:

```
[
    'alt.atheism',
    'comp.graphics',
    'comp.os.ms-windows.misc',
    'comp.sys.ibm.pc.hardware',
    'comp.sys.mac.hardware',
    'comp.windows.x',
    'misc.forsale',
    'rec.autos',
    'rec.motorcycles',
    'rec.sport.baseball',
    'rec.sport.hockey',
    'sci.crypt',
    'sci.electronics',
    'sci.med',
    'sci.space',
    'soc.religion.christian',
    'talk.politics.guns',
    'talk.politics.mideast',
```

```
        'talk.politics.misc',
        'talk.religion.misc'
    ]
```

For simplicity, let's consider only four categories, and fetch the training and testing data set:

```
my_categories = ['rec.sport.baseball','rec.motorcycles','sci.space','comp.graphics']
train = fetch_20newsgroups(subset='train', categories=my_categories)
test = fetch_20newsgroups(subset='test', categories=my_categories)
```

You can see the length of the training and test data, and a training data sample by printing:

```
print(len(train.data))
print(len(test.data))
print(train.data[9])
```

Output:

```
2372
1578
From: degroff@netcom.com (21012d)
Subject: Re: Atlas revisited
Organization: Netcom Online Communications Services (408-241-9760 login: guest)
Distribution: sci
Lines: 8

  I found it very interesting that Atlas depended on pressure to
maintain tank geometry....leads me to the question: ? have any
of the SSTO concepts explored pressurized tankage such that the
launch configuration would be significantly different from the
reentry one?  I have long been facinated by pnumatic structures
as conceived and built by Frei Otto and others, a "ballon" tank
SSTO sounds very clever.
```

In order to use this data for machine learning, we need to be able to convert the content of each string (each document) into a vector of numbers. Let's compute first the sparse tf-idf matrix of the training data set:

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer()

X_train_counts=cv.fit_transform(train.data)

from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer = TfidfTransformer()

X_train_tfidf=tfidf_transformer.fit_transform(X_train_counts)
```

Once we have computed the TF-IDF matrix (called X_train_tfidf), we can pass it to the multinomial naive Bayes classifier to create a predictive model:

```
from sklearn.naive_bayes import MultinomialNB

model = MultinomialNB().fit(X_train_tfidf, train.target)
```

We can apply the model:

```
docs_new = ['Pierangelo is a really good baseball player','Maria rides her motorcycl
e', 'OpenGL on the GPU is fast', 'Pierangelo rides his motorcycle and goes to play f
ootball since he is a good football player too.']

X_new_counts = cv.transform(docs_new)

X_new_tfidf = tfidf_transformer.transform(X_new_counts)

predicted = model.predict(X_new_tfidf)

for doc, category in zip(docs_new, predicted):

    print('%r => %s' % (doc, train.target_names[category]))
```

Output:

```
'Pierangelo is a really good baseball player' => rec.sport.baseball

'Maria rides her motorcycle' => rec.motorcycles

'OpenGL on the GPU is fast' => comp.graphics

'Pierangelo rides his motorcycle and goes to play football since he plays football too.'

      => rec.sport.baseball
```