4. Text data structure

- frequency vector
- subject vector
- TF-IDF

Sogang University Department of Business Administration
Professor Myung Suk Kim

Word expression

◆ word vectors (word representations)

- The most basic problem of natural language processing is how to make a computer recognize natural language.
- Computer recognizes natural language as binary code (Unicode , ASCII code ,...).
  Ex) English : 1100010110111000, English : 1100010110110100
- This way of expression has no characteristics of words at all.
- It can be used for classification and clustering.

◆ One-Hot encoding (one-hot encoding)

- words It is expressed as a single vector , with only one 1 at a specific position, and the rest are marked as 0.

Ex) {Thomas Jepperson made Jepperson building}

	Thomas	Jepperson	made	building
Thomas	One	0	0	0
Jepperson	0	One	0	0
made	0	0	One	0
Jepperson	0	One	0	0
building	0	0	0	One

- To know what the nth word is in a row (sentence) , you need to know the column value that is 1 in that row. (100% restorable)
- One row is each binary row vector , where only one is 1 and the rest are 0.
- Columns act as a dictionary of words (terms).

Word expression

◆ One-hot Disadvantages and Alternatives

- one - hot The disadvantage of encoding is that it becomes inefficient because the size of the vector becomes large when there are many words.
- To overcome various disadvantages, two alternatives are proposed.

◆ Two alternatives

(1) frequency information based

(i) word frequency vector (bag of words)
(ii) word - document matrix method (TF-IDF , etc.)
(iii) co-occurrence matrix : word - word matrix , document - document matrix

(2) Meaning (subject / characteristic) information-based

(i) subject vector (semantic vector)
(ii) word2vec/ Glove (method is different, but the solution is the same)
(iii) BERT, GPT, ...

◆ Word frequency vector (word collection vector): Bag of words

- A method of trying to understand the meaning of a sentence only with word collection data (word frequency), ignoring the order and grammar of words in a sentence
- Unlike the one-hot vector, it contains only the number of appearances, so it is difficult to reproduce the document .
- In the most basic way, there is a vector of binary word collections .
- Binary word collection vectors are useful for document search indexing, which tells which word is used in which document .

Integer Encoding & Padding

◆ Integer encoding

- It is the basic step among several techniques for converting text to numbers in natural language processing.
- A preprocessing task that maps each word to a unique integer.
- If there are 5,000 words in the text, unique integers mapped to words from 1 to 5,000 in each of the 5,000 words, In other words, an index is given, <u>usually after sorting by word frequency</u>.
- One of the ways to assign integers to words is to create a set of words (vocabulary) in which words are sorted in order of frequency.
  There is a way to assign integers from lowest to highest in order .

◆ padding

- Each sentence (or document) can be of different lengths , but the machine divides all documents of the same length into one matrix. Reports can be grouped together and processed .
- Arbitrarily equalizing the length of several sentences for parallel operation

Word - document procession

- word Frequency (Term Frequency: TF)
  - the number of times the word appeared in the document
  - If a particular word appears frequently in a particular document, the word is said to be closely related to that document.

Ex)

Doc1: the fox chases the rabbit
Doc2: the rabbit ate the cabbage
Doc3: the fox caught the rabbit

Rows are words , columns are documents (TDM: Term-Document Matrix)

	Doc1	Doc2	Doc3
the	2	2	2
fox	1	0	1
rabbit	1	1	1
chases	1	0	0
caught	0	0	1
cabbage	0	1	0
ate	0	1	0

Word – document matrix

- word frequency reverse document frequency
- Zipf's Law : The frequency of use of any word is inversely proportional to the rank of that word. (Ex: 1st place is 3 times as frequent as 3rd place)
- Give low weight to words that appear frequently in the document but do not help to understand the meaning of the document -> IDF
- IDF: A weight that measures the importance of a word.
  IDF = log(N/DF). N is the total number of documents.
  DF is the document frequency (the number of documents in which the word appears)
- The smaller the DF, the higher the importance of the word.
- The higher the IDF, the higher the importance of the word
- Words with high TF–IDF values give high discrimination in documents (important words in information retrieval)
- Calculate TF-IDF : (TF-IDF)(t, d)=TF(t, d) x IDF(t), where t is the word and d is the document

	Doc1	Doc2	Doc3	DF	N/DF	IDF=$\log_2(N/DF)$
the	2	2	2	3	3/3	$\log_2(3/3)$
fox	1	0	1	2	3/2	$\log_2(3/2)$
rabbit	1	1	1	3	3/3	$\log_2(3/3)$
chases	1	0	0	1	3/1	$\log_2(3/1)$
caught	0	0	1	1	3/1	$\log_2(3/1)$
cabbage	0	1	0	1	3/1	$\log_2(3/1)$
ate	0	1	0	1	3/1	$\log_2(3/1)$

Word – document matrix

◆ TF standardization and regularization

- The longer the length of the document, the higher the frequency of occurrence of the word and the higher the possibility of being searched.
- Thus the longer the length of the document, the higher the possibility of similarity with other documents .
- Standardization and normalization of TF is necessary to complicate these week-points.

◆ Standardization : z = [TF-mean(TF)]/ standard deviation (TF)

Example) doc 1: mean(TF)=5/7 (number of occurrences / total number of words)

standard deviation (TF)= $\sqrt{\{(2-mean(TF))^2+3(1-mean(TF))^2+3(0-mean(TF))^2\}/6}$

	Doc1	Doc2	Doc3
the	1.70084	1.70084	1.70084
fox	0.37796	-0.944911	0.37796
rabbit	0.37796	0.37796	0.37796
chases	0.37796	-0.944911	-0.944911
caught	-0.944911	-0.944911	0.37796
cabbage	-0.944911	0.37796	-0.944911
ate	-0.944911	0.37796	-0.944911

Word - document procession

◆ Normalization

- divide TF by the total frequency of the word (1+log(TF))/ ni

  ni : frequency count of total words in

	Doc1	Doc2	Doc3
the	0.4 = $(1+\log_2 2)/5$	0.4	0.4
fox	0.2 = $(1+\log_2 1)/5$	0	0.2
rabbit	0.2	0.2	0.2
chases	0.2	0	0
caught	0	0	0.2
cabbage	0	0.2	0
ate	0	0.2	0

Word - document procession

◆ Normalized TF-IDF: Normalized TF times IDF

	정규화 TF-IDF		
	Doc1	Doc2	Doc3
the	0 = 0.4x$\log_2(3/3)$	0=0.4x$\log_2(3/3)$	0
fox	0.11699 = 0.2x$\log_2(3/2)$	0	0.11699
rabbit	0=0.2x$\log_2(3/3)$	0	0
chases	0.31699=0.2x$\log_2(3/1)$	0	0
caught	0	0	0.31699
cabbage	0	0.31699	0
ate	0	0.31699	0

◆ Disadvantages of

- Vectors of two pieces of text with different words, even though they have similar meanings (subjects), are in the TF-IDF vector space.
  If words with the same meaning but different spellings, TF-IDF vectors do not lie close together in the vector space.
  The TF-IDF method is difficult to use in the process of finding documents that are similar in meaning (topic) .

Co-occurrence matrix

◆ joint (simultaneous) occurrence matrix

A method of directly counting the number of times words appear simultaneously in a particular context . The number of simultaneous appearances is expressed as a matrix and the matrix is digitized to create word vectors .

Ex) Myeong-seok and Jun-seon went to America
Myeong-seok and Sang-ho went to the library
Myeong-seok and Jun-seon like cold noodles .

< Co-occurrence matrix : word - word matrix > → square matrix , symmetric matrix

	Myeongseokgwa	Joon Seon Eun	trade name	to the USA	to the library	cold noodles	went	I like it .
Myeongseokgwa	0	2	One	One	One	One	2	One
Joon Seon Eun	2	0	0	One	0	One	One	One
trade name	One	0	0	0	One	0	One	0
to the USA	One	One	0	0	0	0	One	0
to the library	One	0	One	0	0	0	One	0
cold noodles	One	One	One	0	0	0	0	One
went	2	One	One	One	One	0	0	0
I like it .	One	One	One	0	0	One	0	0

➡ Can be used as a social network analysis (e.g., calculating the centrality of each word (degree of co nnection , proximity , median , eigenvector))

Word embedding

◆ Word embedding (word embedding)

- A one-hot vector is a sparse representation with many 0 's and only one 1'.
- In contrast to sparse representation , the size of the vector is determined by a value set by the user (smaller than the size of the word set) rather than the size of the word set , and has real values other than 0 and 1.
- The method of expressing words as dense vectors is called word embedding, and the result obtained in this way is called an embedding vector.
- Examples of word embeddings include, LSA, word2vec, FastText , and Glove.

◆ Distributed representation

- Local representation is a method of expressing a word by looking only at the word itself and mapping a specific value.
- On the other hand , the distributed representation depends on the distribution hypothesis
- Based on the expression, it is made on the assumption that words appearing in similar positions have similar meanings, and the task of vectorizing the similarity of words corresponds to word embedding.
- Distributed representation methods refer to neighboring words to represent that word.
- For example, since the words cute and lovely often appear near the word puppy, the word puppy defines the word as cute and lovely.
- As an example of a distributed representation, There are techniques such as Word2vec.

https://wikidocs.net/31767

Topic vector

- **Topic vector (semantic vector):**

- Dimensional reduction of multidimensional vectors whose components are subject scores obtained using the weighted frequencies of TF-IDF vectors .
- Group words of the same subject together using correlations between normalized term frequencies .
- Used for semantic-based retrieval, which searches documents based on their semantics . -> usually than keyword-based search is known to be accurate .
- Able to find a set of key words (keywords) that best summarize the meaning of a given document .
- There are (1) word subject vectors representing the meaning of words and (2) document subject vectors representing the meaning of documents .

▪ Word Topic Vectors : Create 3 topic scores {pet}, {animal } , {city} as subject vector reproduce

(1) given TF-IDF vector

cat	dog	apple	lion	NYC	love
1.1733	1.5322	0.3211	0.8774	2.4432	0.2727

(2) generate subject vectors by weighting the TF-IDF vectors (associativity Randomly assigning high weights to high words)

pet = 0.3 xcat+ 0.3 xdog+0.0xapple+0.0xlion-0.2xNYC+ 0.2 xlove → {0.3, 0.3, 0.0, 0.0, -0.2, 0.2}
animal = 0.1 xcat+ 0.1 xdog-0.1xapple+0.5xlion+0.1xNYC +0.1 xlove → {0.1, 0.1, -0.1, 0.5, 0.1, 0.1}
city = 0.0 xcat -0.1 xdog+0.2xapple-0.1xlion+0.5xNYC+ 0.1 xlove → {0.0, -0.1, 0.2, -0.1, 0.5, 0.1}

(3) Creating a 6 dimensional word topic vector using 3 topic scores : It is possible to determine the degree of similarity

cat = 0.3 xpet+ 0.1 xanimal+ 0.0 xcity → {0.3, 0.1, 0.0}
dog = 0.3 xpet+ 0.1 xanimal -0.1 xcity → {0.3, 0.1, 0.1}
...
love = 0.2 xpet+ 0.1 xanimal+ 0.1 xcity → {0.2, 0.1, 0.1}

each Represent words as subject vectors

Topic vector

- **Document subject vector**

  To obtain a word2vec that contains the topic (meaning) of the entire document, the document topic vector is obtained as the sum of word vectors.

- **Word inference using subject vectors**

  It can be converted to a word vector space with a lower dimension than the word frequency vector, and the word vector operation has meaning .

  Useful for word analogy tasks .

  Ex) king : male = female : ?

➢ Topic : { male / female , adult / child , royal family / commoner }

➢ Words : King = { 1.0, 0.9, 0.9 }, Prince = {0.9, 0.1, 0.8}, Queen = { 0.1, 0.9, 0.8 }, Princess = {0.1, 0.1, 0.8}

male = { 1.0, 0.0, 0.0 }, female = { 0.1, 0.0, 0.0 }

king - male + female = { 0.1, 0.9, 0.9 } → close to queen

Topic Vector Expansion : Word2vec and BERT

◆ Word2vec (Tomasi Mikolov , MS Apprentice , 2012)

- Subject vectors are the same If it is only in a sentence, it has meaning as a word, but word2vec has meaning as a word nearby .
- n-gram consisting of words before and after the target word
- Use window=k to specify
- CBOW (Continuous Bag of Words) and Skip-Gram are available

◆ BERT: Bidirectional Encoder Representations from Transformer (Google , 2018): Encoder only model

- Train the model using the encoder part of Pre-learning is performed using two language learning methods: mask language model and next sentence prediction.

◆ Difference between Word2vec and BERT

- Word2vec corresponds to the static embedding technique , so multiple meanings corresponding to one word are converted into only one vector.
- A fixed expression and has the same expression value wherever it appears in the document text (regardless of order), so homophones cannot be distinguished .
- As a solution to this, contextual embedding creates a dynamic vector for each context based on the sentence. The technique BERT, ELMo etc. are developed to solve this problem.

Integer Encoding

Integer Encoding

1) Dictionary

```
# https://wikidocs.net/31766
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import nltk
nltk.download('punkt')
nltk.download('stopwords’)

raw_text = "A barber is a person. a barber is good person. a barber is huge person. he Knew A Secret! The Secret He Kept is huge secret. Huge secret. His barber kept his word. a barber kept his word. His barber kept his secret. But keeping and keeping such a huge secret to himself was driving the barber crazy. the barber went up a huge mountain. "

# Tokenizing sentence
sentences = sent_tokenize(raw_text)
print(sentences)
```

Integer Encoding

```
vocab = {}
preprocessed_sentences = []
stop_words = set(stopwords.words('english'))

for sentence in sentences:  # Tokenizing words
   tokenized_sentence = word_tokenize(sentence)
   result = []

for word in tokenized_sentence :
word = word.lower () # Reduce the number of words by lowercase all words .
if word not in stop_words : # Remove stopwords for word tokenized results .
if len (word) > 2: # Remove additional words for word length less than or equal to 2 .
            result. append (word)
if word not in vocab:
vocab[word] = 0
vocab[word] += 1
   preprocessed_sentences. append (result)
print( preprocessed_sentences )

print(' word set :',vocab)
# print the frequency of the word 'barber'
print(vocab["barber"])
```

Integer Encoding

```
# Build a dictionary based on frequencies
vocab_sorted = sorted( vocab.items (), key = lambda x:x[1], reverse = True)
print( vocab_sorted )

word_to_index = {}
i = 0
for (word, frequency) in vocab_sorted:
  if frequency > 1 : # Exclude low frequency words
    i = i + 1
    word_to_index [word] = i    # Index words based on frequency
print( word_to_index )


# Remove words with index greater than 5 ( remove low frequency words )
vocab_size = 5
words_frequency = [word for word, index in word_to_index.items () if index >= vocab_size + 1]

# Delete the index information for the word
for w in words_frequency :
del word_to_index [w]
print( word_to_index )

# Words with an index greater than 5 ( low frequency ) are collectively referred to as
word_to_index ['OOV'] = len ( word_to_index ) + 1
print( word_to_index )
```

Integer Encoding

```
encoded_sentences = []
for sentence in preprocessed_sentences :
   encoded_sentence = []
for word in sentence:
try:
         # If a word is in the word set, return the integer for that word
         encoded_sentence.append ( word_to_index [word])
except KeyError :
         # If the word is not in the word set, return an integer of
         encoded_sentence.append ( word_to_index ['OOV'])
   encoded_sentences.append ( encoded_sentence )
print( encoded_sentences )
```

2) Counting

```
from collections import Counter
print( preprocessed_sentences )

# words = np.hstack ( preprocessed_sentences ) can also be done
all_words_list = sum( preprocessed_sentences , [])
print( all_words_list )

# Python's Count the frequency of words using the Counter module
vocab = Counter( all_words_list )
print(vocab)

print(vocab["barber"]) # print the frequency of the word 'barber'
```

Integer Encoding

```
vocab_size = 5
vocab = vocab.most_common ( vocab_size ) # Store only the top 5 most frequent words

word_to_index = {}
i = 0
for (word, frequency) in vocab:
   i = i + 1
   word_to_index [word] = i
print( word_to_index )
```

3) NLTK 's FreqDist

```
from nltk import FreqDist
import numpy as np

# Remove sentence breaks with np.hstack
vocab = FreqDist ( np.hstack ( preprocessed_sentences ))
print(vocab["barber"]) # print the frequency of the word 'barber'

vocab_size = 5
vocab = vocab.most_common ( vocab_size ) # Store only the top 5 most frequent words
print(vocab)

word_to_index = {word[0] : index + 1 for index, word in enumerate(vocab)}
print( word_to_index )
```

Integer Encoding

4) Understanding enumerate

```
test_input = ['a', 'b', 'c', 'd', 'e']
for index, value in enumerate( test_input ): # Index from
print("value : {}, index: {}".format(value, index))
```

5) Keras Text Preprocessing

```
from tensorflow.keras.preprocessing.text import Tokenizer
preprocessed_sentences = [['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'] ,
['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept ', 'word'], ['barber', 'kept', 'secret'],
['keeping', 'keeping', 'huge', 'secret', 'driving', 'barber', 'crazy'] , ['barber', 'went', 'huge', 'mountain']]

tokenizer = Tokenizer()

# Input corpus into
tokenizer.fit_on_texts(preprocessed_sentences)
print(tokenizer.word_index)
print(tokenizer.word_counts)
print(tokenizer.texts_to_sequences(preprocessed_sentences))

vocab_size = 5
tokenizer = Tokenizer(num_words = vocab_size + 1) # 상위 5개 단어만 사용
tokenizer.fit_on_texts(preprocessed_sentences)
print ( tokenizer . word_index )
print ( tokenizer . word_counts )
print( tokenizer . texts_to_sequences ( preprocessed_sentences ));
```

Configuration (Integer Encoding)

```
vocab_size = 5
words_frequency = [ word for word , index in tokenizer . word_index . items ( ) if index >= vocab_size +

# delete cases with more than 5 ubdexes
for word in words_frequency :
del tokenizer.word_index [word] # delete index information for that word
   del tokenizer.word_counts [word] # delete count information for that word

print( tokenizer. word_index )
print( tokenizer. word_counts )
print( tokenizer. texts_to_sequences ( preprocessed_sentences ))



# Size of word set is +2 , taking into account the number 0 and OOV
vocab_size = 5
tokenizer = Tokenizer( num_words = vocab_size + 2, oov_token = 'OOV');
tokenizer.fit_on_texts ( preprocessed_sentences );

print(' Token OOV value : {}'.format( tokenizer.word_index ['OOV']))
print( tokenizer . texts_to_sequences ( preprocessed_sentences ));
```

Padding

1. Padding using Numpy

```
# https://wikidocs.net/83544
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer

preprocessed_sentences = [['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], ['keeping', 'keeping', 'huge', 'secret', 'driving', 'barber', 'crazy'], ['barber', 'went', 'huge', 'mountain’]]

tokenizer = Tokenizer()
tokenizer.fit_on_texts(preprocessed_sentences)
encoded = tokenizer.texts_to_sequences(preprocessed_sentences)
print(encoded)

max_len = max(len(item) for item in encoded)
print('최대 길이 :',max_len)


for sentence in encoded:
   while len(sentence) < max_len:
      sentence.append(0)
padded_np = np.array(encoded)
padded_np
```

Padding

2. Padding using Keras tools

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
encoded = tokenizer.texts_to_sequences(preprocessed_sentences)
print(encoded)

padded = pad_sequences(encoded)
padded = pad_sequences(encoded, padding='post’)
(padded == padded_np).all()

padded = pad_sequences (encoded, padding='post', maxlen =5)
padded = pad_sequences (encoded, padding='post', truncating='post', maxlen =5)

last_value = len ( tokenizer.word_index ) + 1 # use a number one greater than the size of the word set
print( last_value )

padded = pad_sequences (encoded, padding='post', value= last_value )
```

Word expression

Creating

1) str.split (sentence) : split into tokens
2) sorted (set(str.split (sentence))): Creating a vocabulary
3) '.'.join(vocab): Sort tokens in ASCII order (digits first / uppercase letters first)
4) np.zeros ((number of rows , number of columns), int): Create a zero matrix with rows (number of original words) and columns (number of words in dictionary) (numpy =np package)
5) for i , word in enumerate(token_sequence):

   onehot_vectors [i , vocab.index (word)] = 1

   One-hot vector generation
6) df = pd.DataFrame (onehot_vectors , columns=vocab): as dataframe
   Create and assign word names to columns
   (pandas = pd package)
7) df [df == 0] = '' : remove zero from matrix

the nth word is , the You just need to know the column value that is 1 in the row .
Single A row is each binary row vector, where only one is 1 and the others are 0 .

Word expression

1. One-hot vector creation

```
import numpy as np
sentence = "Thomas Jefferson began building Jefferson Monticello at the age of 26."
sentence.split ()
token_sequence = str.split (sentence) # split the sentence into tokens
vocab = sorted (set( token_sequence )) # Create token lexicon : remove duplicate words
'.'.join(vocab) # Sort tokens in ASCII order ( numbers first / caps first )
num_tokens = len ( token_sequence )
vocab_size = len (vocab)
onehot_vectors = np.zeros (( num_tokens , vocab_size ), int )
# create zero matrix with rows ( original words ) and columns ( dictionary words )
for i , word in enumerate( token_sequence ):
    onehot_vectors [ i , vocab.index (word)] = 1
'.'.join(vocab)
onehot_vectors

import pandas as pd
df = pd.DataFrame ( onehot_vectors , columns=vocab) # make it a dataframe and give the columns word names
print( df )
df [ df == 0] = '' # remove zero
print( df )
```

Word expression

2. Binary word collection vectors to make

(1) In case of one sentence

(2) In case of multiple sentences

(3) dot product calculation

```
import numpy as np
v1 = np.array ([1, 2, 3])
v2 = np.array ([2, 3, 4])
v1.dot(v2) # dot product method 1
(v1*v2).sum() # dot product method 2
sum([x1*x2 for x1, x2 in zip(v1, v2)]) # dot product method 3
```

(4) Counting the number of duplicate words in two word collection vectors

https://wikidocs.net/22647

Word expression

2. A collection of binary word vectors

```
# (1) In the case of one sentence
sentence = "Thomas Jefferson began building Jefferson Monticello at the age of 26."
sentence_bow = {}
for token in sentence. split ():
    sentence_bow [token] = 1
sorted( sentence_bow. items ())

# Assign the number 1 to each word and make it into a data frame (T is transpose)
import pandas as pd
df = pd.DataFrame(pd.Series(dict([(token, 1) for token in sentence.split()])), columns=['sent']).T
df


# (2) several sentences case
sentences = "Thomas Jefferson began building Jefferson Monticello at the age of 26.₩n"₩
            "Construction was done mostly by local masons and carpenters.₩n" ₩
    "He moved into the South Pavilion in 1770.₩n" ₩
    "Turning Monticello into a neoclassical masterpiece in the Palladian style was his perennial project.₩n"

corpus = {}
for i , sent in enumerate( sentences.split ('₩n')): # ₩n is the sentence split criterion
   corpus['sent{}'.format( i )] = dict ([(token, 1) for token in sent.split ()])

df1 = pd.DataFrame.from_records (corpus). fillna (0). astype ( int ).T
df1[df1.columns[:10]] # print only the first 10 tokens
```

Word expression

```
#(3) dot product calculation
import numpy as np
v1 = np.array ([1, 2, 3])
v2 = np.array ([2, 3, 4])
v1.dot(v2) # dot product method 1
(v1*v2).sum() # dot product method 2
sum([x1*x2 for x1, x2 in zip(v1, v2)]) # dot product method 3


#(4) Count the number of duplicate words in two word collection vectors (T is transpose)
#sent0 = "Thomas Jefferson began building Jefferson Monticello at the age of 26."
#sent1 = "Construction was done mostly by local masons and carpenters."
#sent2 = "He moved into the South Pavilion in 1770."
#sent3 = "Turning Monticello into a neoclassical masterpiece in the Palladian style was his perennial project."
df2 = df1.T
df2.sent0.dot(df2.sent1)
df2.sent0.dot(df2.sent2)
df2.sent0.dot(df2.sent3)
[(k, v) for (k, v) in (df2.sent0 & df2.sent3).items() if v]
```

Word expression

3. bag_of_words
(1) Sentence Extract only unique words within (create a dictionary)

```
From nltk.tokenize import TreebankWordTokenizer
TreebankWordTokenizer.tokenize ( sentence.lower ())
```

(2) Count the number of words in a sentence

```
from collections import Counter
bag_of_words = Counter(tokens)
```

(3) TF calculation

```
freq_harry = bag_of_words ['harry']
num_unique = len ( bag_of_words )
tf = freq_harry / num_unique
```

4. Making a matrix of the frequencies of

```
from collections import Counter
term = ['faster', 'Harry', 'Jill', 'home']
vector1 = Counter( tok for tok in tokenize("The faster Harry got to the store.") if tok in lexicon)
```

Word expression

3. Create a Bag of words

```
# (1) Extract words in a sentence
from nltk.tokenize import TreebankWordTokenizer
sentence = "The faster Harry got to the store, the faster and faster Harry would get home."
tokenizer = TreebankWordTokenizer ()
tokenize = tokenizer. tokenize
tokens = tokenize( sentence. lower ())

# (2) count the number of words in a sentence
from collections import Counter
bag_of_words = Counter(tokens)

# Extract top 2 words based on frequency
bag_of_words.most_common (3)

# Extract word frequency
freq_harry = bag_of_words ['harry']

# number of unique tokens in sentence
num_unique = len ( bag_of_words )

# (3) TF (term-frequency) calculation
tf = freq_harry / num_unique
round( tf , 4) # round to 4 decimal places
```

Word expression

4. Create a matrix of word frequencies representing multiple documents

```
term = ['faster', 'Harry', 'Jill', 'home']
bag_of_words1 = Counter( tok for tok in tokenize("The faster Harry got to the store, the faster and faster Harry would get home.") if tok in term)
bag_of_words2 = Counter(tok for tok in tokenize("Jill is faster than Harry.") if tok in term)
bag_of_words3 = Counter(tok for tok in tokenize("Jill and Harry fast.") if tok in term)
corpus = [bag_of_words1, bag_of_words2, bag_of_words3]

import pandas as pd
df = pd.DataFrame.from_records(corpus)
df = df.fillna(0)
df
df[:1]
df [1:2]
df [2:3]
```

Word expression

5. Normalized representation of the document Create word vectors

```
document_vector = []
doc_length = len (tokens)
for key, value in kite_count.most_common ():
    document_vector.append (value / doc_length )
```

6. Normalized representation of multiple documents Word vector build function

```
vector_template = OrderedDict((token, 0) for token in lexicon)
document_vectors = []
for doc in [doc_0, doc_1, doc_2]:
    vec = copy.copy(vector_template)  # So we are dealing with new objects, not multiple references
to the same object
    tokens = tokenizer.tokenize(doc.lower())
    token_counts = Counter(tokens)
    for key, value in token_counts.items():
        vec [key] = value/ len (lexicon)
    document_vectors.append ( vec )
```

Word expression

5. Building normalized term vectors

kite_text = "A kite is traditionally a tethered heavier-than-
air craft with wing surfaces that react against the air to create lift and drag. A kite consists of wings, tethers, and anch
ors. Kites often have a bridle to guide the face of the kite at the correct angle so the wind can lift it. A kite's wing also
 may be so designed so a bridle is not needed; when kiting a sailplane for launch, the tether meets the wing at a sing
le point. A kite may have fixed or moving anchors. Untraditionally in technical kiting, a kite consists of tether-set-
coupled wing sets; even in technical kiting, though, a wing in the system is still often called the kite. The lift that susta
ins the kite in flight is generated when air flows around the kite's surface, producing low pressure above and high pre
ssure below the wings. The interaction with the wind also generates horizontal drag along the direction of the wind. T
he resultant force vector from the lift and drag force components is opposed by the tension of one or more of the lin
es or tethers to which the kite is attached. The anchor point of the kite line may be static or moving (e.g., the towing
of a kite by a running person, boat, free-
falling anchors as in paragliders and fugitive parakites or vehicle). The same principles of fluid flow apply in liquids an
d kites are also used under water. A hybrid tethered craft comprising both a lighter-than-
air balloon as well as a kite lifting surface is called a kytoon. Kites have a long and varied history and many different t
ypes are flown individually and at festivals worldwide. Kites may be flown for recreation, art or other practical uses. Sp
ort kites can be flown in aerial ballet, sometimes as part of a competition. Power kites are multi-
line steerable kites designed to generate large forces which can be used to power activities such as kite surfing, kite la
ndboarding, kite fishing, kite buggying and a new trend snow kiting. Even Man-lifting kites have been made."

Word expression

```
from collections import Counter
from nltk.tokenize import TreebankWordTokenizer

# (1) Counting the number of unique words
tokenizer = TreebankWordTokenizer ()
# kite_text = "A kite is traditionally ..." # Step left to user, so we're not repeating ourselves
tokens = tokenizer.tokenize ( kite_text.lower ())
len (tokens)

# (2) Word frequency
token_counts = Counter(tokens)
print( token_counts )

# (3) Remove stop words
import nltk
nltk.download (' stopwords ')
stopwords = nltk.corpus.stopwords.words (' english ')

tokens = [x for x in tokens if x not in stopwords ]
kite_count = Counter(tokens)
print( kite_count )

# (4) Create a normalized term frequency vector
document_vector = []
doc_length = len (tokens)
for key, value in kite_count.most_common ():
    document_vector.append (value/ doc_length );
print ( document_vector )
```

Word expression

6. Regularized word vectors describing several sentences

```
doc_0 = "There's nothing like a smile on your face. There's nothing like a smile on your face.
doc_1 = "Harry is hairy and faster than Jill."
doc_2 = "Jill is not as hairy as Harry."

tokens_0 = tokenizer.tokenize (doc_0.lower());
tokens_1 = tokenizer.tokenize (doc_1.lower());
tokens_2 = tokenizer.tokenize (doc_2.lower());
lexicon = sorted(set(tokens_0 + tokens_1 + tokens_2))

from collections import OrderedDict
# zero vector generation
vector_template = OrderedDict ((token, 0) for token in lexicon)

print( vector_template )
# renewing zero vector
import copy
document_vectors = []
for doc in [doc_0, doc_1, doc_2]:
    vec = copy.copy(vector_template)  #So we are dealing with new objects, not multiple references to the same object
    tokens = tokenizer.tokenize(doc.lower())
    token_counts = Counter(tokens)
    for key, value in token_counts.items():
        vec[key] = value / len(lexicon)
    document_vectors.append ( vec )

print( document_vectors )
```

Document word matrix

1. Zipf's Law

The frequency of the first word is 2 times the frequency of the second word, three times the frequency of the third word ...

2.TF

(1) Comparison of the number of occurrences of the word

(2) Comparison of the number of occurrences of the word and --> It occurs frequently, but it cannot be sai d to be important . IDF concept required

3.IDF

(1) Count the number of documents with and, kite, and China in the document

(2) Finding

(3) IDF calculation

4.TF-IDF

5. TF-IDF obtained using multiple documents

6. Creating

7. Corpus to Scikit Learn Create

Document word matrix

1. Zipf's Law

```
import nltk
from nltk.corpus import brown
nltk.download ('brown')
brown.words ()[0:10]
print( len ( brown. words ()))
brown.tagged_words()[:5]

# most frequent 20 words extraction
from collections import Counter
puncs = [',', '.', '--', '-', '!', '?', ':', ';', '``', "''", '(', ')', '[', ']']
word_list = [x.lower() for x in brown.words() if x not in puncs]
token_counts = Counter(word_list)
print(token_counts.most_common(20))
```

Document word matrix

2. Compare words in two documents

```
from nltk.tokenize import TreebankWordTokenizer
tokenizer = TreebankWordTokenizer ()

kite_text0 = "A kite is traditionally a tethered heavier-than-
air craft with wing surfaces that react against the air to create lift and drag. A kite consists of wings, tethers, and anch
ors. Kites often have a bridle to guide the face of the kite at the correct angle so the wind can lift it. A kite's wing also
 may be so designed so a bridle is not needed; when kiting a sailplane for launch, the tether meets the wing at a sing
le point. A kite may have fixed or moving anchors. Untraditionally in technical kiting, a kite consists of tether-set-
coupled wing sets; even in technical kiting, though, a wing in the system is still often called the kite. The lift that susta
ins the kite in flight is generated when air flows around the kite's surface, producing low pressure above and high pre
ssure below the wings. The interaction with the wind also generates horizontal drag along the direction of the wind. T
he resultant force vector from the lift and drag force components is opposed by the tension of one or more of the lin
es or tethers to which the kite is attached. The anchor point of the kite line may be static or moving (e.g., the towing
of a kite by a running person, boat, free-
falling anchors as in paragliders and fugitive parakites or vehicle). The same principles of fluid flow apply in liquids an
d kites are also used under water. A hybrid tethered craft comprising both a lighter-than-
air balloon as well as a kite lifting surface is called a kytoon. Kites have a long and varied history and many different t
ypes are flown individually and at festivals worldwide. Kites may be flown for recreation, art or other practical uses. Sp
ort kites can be flown in aerial ballet, sometimes as part of a competition. Power kites are multi-
line steerable kites designed to generate large forces which can be used to power activities such as kite surfing, kite la
ndboarding, kite fishing, kite buggying and a new trend snow kiting. Even Man-lifting kites have been made."
```

Document word matrix

```
history_text0 = 'Kites were invented in China, where materials ideal for kite building were readily available: silk fabric f
or sail material; fine, high-tensile-
strength silk for flying line; and resilient bamboo for a strong, lightweight framework. The kite has been claimed as th
e invention of the 5th-
century BC Chinese philosophers Mozi (also Mo Di) and Lu Ban (also Gongshu Ban). By 549 AD paper kites were certai
nly being flown, as it was recorded that in that year a paper kite was used as a message for a rescue mission. Ancient
and medieval Chinese sources describe kites being used for measuring distances, testing the wind, lifting men, signali
ng, and communication for military operations. The earliest known Chinese kites were flat (not bowed) and often recta
ngular. Later, tailless kites incorporated a stabilizing bowline. Kites were decorated with mythological motifs and legen
dary figures; some were fitted with strings and whistles to make musical sounds while flying. From China, kites were in
troduced to Cambodia, Thailand, India, Japan, Korea and the western world. After its introduction into India, the kite fu
rther evolved into the fighter kite, known as the patang in India, where thousands are flown every year on festivals su
ch as Makar Sankranti. Kites were known throughout Polynesia, as far as New Zealand, with the assumption being that
 the knowledge diffused from China along with the people. Anthropomorphic kites made from cloth and wood were u
sed in religious ceremonies to send prayers to the gods. Polynesian kite traditions are used by anthropologists get an
idea of early "primitive" Asian traditions that are believed to have at one time existed in Asia.'

intro_text = kite_text0.lower();
intro_tokens = tokenizer.tokenize ( intro_text );
history_text = history_text0.lower()
history_tokens = tokenizer.tokenize ( history_text );
intro_total = len ( intro_tokens )
history_total = len ( history_tokens )
```

Document word matrix

```
# (1) Compare the number of occurrences of the word kite in history and kite documents
intro_tf = {} # dictionary object
history_tf = {} # dictionary object
intro_counts = Counter( intro_tokens )
intro_tf['kite'] = intro_counts['kite'] / intro_total
history_counts = Counter(history_tokens)
history_tf['kite'] = history_counts['kite'] / history_total
print('Term Frequency of "kite" in intro is: {}'.format(intro_tf['kite']))
print('Term Frequency of "kite" in history is: {}'.format(history_tf['kite']))


# (2) Comparison of the number of occurrences of the word and -> It appears a lot, but it is not important.
IDF concept required
intro_tf ['and'] = intro_counts ['and'] / intro_total
history_tf ['and'] = history_counts ['and'] / history_total
print('Term Frequency of "and" in intro is: {}'. format( intro_tf ['and']))
print('Term Frequency of "and" in history is: {}'.format( history_tf ['and']))
```

Document word matrix

3. IDF

```
# (1) Kite and, kite, China
num_docs_containing_and = 0
for doc in [ intro_tokens , history_tokens ]:
if 'and' in doc:
        num_docs_containing_and += 1

num_docs_containing_kite = 0
for doc in [ intro_tokens , history_tokens ]:
if 'kite' in doc:
        num_docs_containing_kite += 1

num_docs_containing_china = 0
for doc in [ intro_tokens , history_tokens ]:
if 'china' in doc:
        num_docs_containing_china += 1


# (2) Finding the TF of China from the two documents
intro_tf ['china'] = intro_counts ['china'] / intro_total
history_tf ['china'] = history_counts ['china'] / history_total
```

Document word matrix

```
# (3) IDF 계산
num_docs = 2
intro_idf = {}
history_idf = {}
intro_idf['and'] = num_docs / num_docs_containing_and
history_idf['and'] = num_docs / num_docs_containing_and
intro_idf['kite'] = num_docs / num_docs_containing_kite
history_idf['kite'] = num_docs / num_docs_containing_kite
intro_idf['china'] = num_docs / num_docs_containing_china
history_idf['china'] = num_docs / num_docs_containing_china
```

4. TF-IDF

```
# kite 문서
intro_tfidf = {}
#intro_tfidf['and'] = intro_tf['and'] * intro_idf['and']
intro_tfidf['kite'] = intro_tf['kite'] * intro_idf['kite']
intro_tfidf['china'] = intro_tf['china'] * intro_idf['china']

# history 문서
history_tfidf = {}
#history_tfidf['and'] = history_tf['and'] * history_idf['and']
history_tfidf['kite'] = history_tf['kite'] * history_idf['kite']
history_tfidf['china'] = history_tf['china'] * history_idf['china']
```

Document word matrix

5. TF-IDF using several documents

```
from nltk.tokenize import TreebankWordTokenizer
sentence = "The faster Harry got to the store, the faster and faster Harry would get home."
tokenizer = TreebankWordTokenizer()

lexicon = ['faster', 'Harry', 'Jill', 'home']
doc_0 = "The faster Harry got to the store, the faster Harry, the faster, would get home."
doc_1 = "Harry is hairy and faster than Jill."
doc_2 = "Jill is not as hairy as Harry."

from collections import OrderedDict
# zero-vector
vector_template = OrderedDict ((token, 0) for token in lexicon)

import copy
document_tfidf_vectors = []
documents = [doc_0, doc_1, doc_2]
documents
```

Document word matrix

```
for doc in documents:

    vec = copy.copy ( vector_template ) # So we are dealing with new objects, not multiple references to the same
object

    tokens = tokenizer.tokenize(doc.lower())
    token_counts = Counter(tokens)

    for key, value in token_counts.items():
        docs_containing_key = 0
        for _doc in documents:
          if key in _doc:
            docs_containing_key += 1
        tf = value / len(lexicon)
        if docs_containing_key:
            idf = len (documents) / docs_containing_key
else:
            idf = 0
        vec [key] = tf * idf
    document_tfidf_vectors.append ( vec )

document_tfidf_vectors
```

Document word matrix

6. To Scikit Learn Create TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [doc_0, doc_1, doc_2]
vectorizer = TfidfVectorizer(min_df=1)
model = vectorizer.fit_transform(corpus)
print(model.todense())
```

7. TF-IDF using sklearn

```
from sklearn.feature_extraction.text import TfidfVectorizer

CORPUS = ['"Hello world!"', 'Go fly a kite.', 'Kite World', 'Take a flying leap!', 'Should I fly home?']
def tfidf_corpus(docs=CORPUS):
    vectorizer = TfidfVectorizer()
    vectorizer = vectorizer.fit(docs)
    return vectorizer, vectorizer.transform(docs)  # (TfidfVectorizer, tfidf_vectors)

tfidf_corpus(CORPUS)
```

Document word matrix

8. Example

```
# https://wikidocs.net/31698
import pandas as pd # for dataframe usage
from math import log # for IDF calculation

docs = [
' The apple I want to eat ',
' I want to eat bananas ',
' long yellow banana banana ',
' I like fruit '
]
vocab = list(set(w for doc in docs for w in doc.split ()))
vocab.sort()

N = len(docs) # total number of documents

def tf(t, d):
    return d.count(t)

def idf(t):
    df = 0
    for doc in docs:
        df += t in doc
    return log(N/(df + 1))

def tfidf(t, d):
return tf ( t,d )* idf (t)
```

Document word matrix

```
##### TF output on DTM
result = []
for i in range(N): # Execute the following command for each document
   result. append ([])
d = docs[ i ]
for j in range( len (vocab)):
t = vocab[j]
result[-1]. append( tf (t, d))

tf_ = pd.DataFrame (result, columns = vocab)
tf_ _


##### IDF for each word
result = []
for j in range( len (vocab)):
t = vocab[j]
    result.append ( idf (t))

idf_ = pd.DataFrame (result, index = vocab, columns = ["IDF"])
idf_ _
```

Document word matrix

```
##### TF-IDF matrix output
result = []
for i in range(N):
    result. append ([])
d = docs[ i ]
for j in range( len (vocab)):
t = vocab[j]

result[-1].append( tfidf ( t,d ))

tfidf_ = pd.DataFrame (result, columns = vocab)
tfidf_ _


##### Creating DTM and TF-IDF through Scikit-Learn
from sklearn.feature_extraction.text import CountVectorizer
corpus = [
'you know I want your love',
'I like you',
'what should I do',
]
vector = CountVectorizer ()
print( vector. fit_transform (corpus). toarray ()) # Record the frequency count of each word from the corpus .
print( vector.vocabulary _) # Show how each word was indexed .
```

Document word matrix

```
##### TfidfVectorizer Use
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [
'you know I want your love',
'I like you',
'what should I do',
]
tfidfv = TfidfVectorizer ().fit(corpus)
print( tfidfv.transform (corpus).toarray ( ))
print( tfidfv. vocabulary _)
```

Applications

1. Data structure using IMDB movie reviews

```
!pip install Afinn

import pandas as pd
import nltk
from afinn import afinn
nltk.download (' stopwords ')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import RegexpTokenizer
import numpy as np
import matplotlib.pyplot as plt

# Download below data from Kaggle
#https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

# Preprocessing command to save file in colab ( set path )
from google.colab import drive
drive.mount('/content/gdrive')

#file name & path
file_name = "/content/gdrive/My Drive/Colab Notebooks/Textmining/download/IMDB Dataset.csv"
review = pd.read_csv(file_name, engine="python")
review.head(10)
len(review)
review['review'][0]
```

Applications

```
#https://duckkkk.com/entry/Kaggle-IMDB-%EA%B0%90%EC%A0%95-%EB%B6%84%EC%84%9D-Part-1
########## Preprocessing ##########
# Install module to remove HTML tags
from bs4 import BeautifulSoup

# Analyze only the first n reviews
n = 100 # there are 50000 total
reviews = []
for row in review['review'][0:n]:
review1 = BeautifulSoup (row, "html5lib"). get_text ()
  reviews. append (review1)

print(reviews) # get
len (reviews)

# Install the module to use regular expressions
import re

# ^: means start , extracts only letters starting with lowercase letters of the alphabet
review_list = []
for row1 in reviews:
review2 = re.sub ('[^a- zA -z]',' ',row1)
review3 =review2.lower() # Convert all to lower case
  review_list.append(review3)

print(review_list)
len(review_list)
```

Applications

```
########## Tokenizing ##########
token_list = []
for row2 in review_list:
  review4 = row2.split()        # Tokenizing
  token_list.append(review4)

print(token_list)
len(token_list)

# remove stopwords using for loop
sentence_words = [w for w in token_list if not w in stopwords.words('english')]
len(sentence_words)
type(sentence_words)

clean_review = []
for sentence in sentence_words:
  s = ' '
  clean_review.append(s.join(sentence))

clean_review
```

Applications

```
########## Vectorize documents by frequency of occurrence of words : CounterVectorizer #################
#### Convert tokens from reviews to features
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline

# Change the parameter value differently from the tutorial
vectorizer = CountVectorizer (analyzer = 'word',
                              tokenizer = None,
                              preprocessor=None,
                              stop_words = None;
                              min_df = 2, # minimum number of documents for token to appear
                              ngram_range =(1, 1), # specify n-gram
                              max_features = 20000)

# Use pipelines to improve speed
pipeline = Pipeline([(' vect ', vectorizer ),])

# vectorize
train_data_features = pipeline . fit_transform ( clean_review )
train_data_features
train_data_features.shape  # rows ( number of documents ) X columns ( number of words )

vocab = vectorizer.get_feature_names_out () # word names
print( len (vocab))
vocab[:30]

import pandas as pd
df = pd.DataFrame ( train_data_features.toarray ())
print( df )
```

Applications

```
#################### Understand the difference between the two methods ####################
##############
from sklearn.feature_extraction.text import CountVectorizer

text = [" Everyone laughs at love once ",
        " Everyone Cries Over Love ",
        " That's love love love "]

count_vec = CountVectorizer ()
m = count_vec.fit_transform (text)
m.toarray ()

##############
from sklearn.feature_extraction.text import CountVectorizer

text = [[' Everyone ',' Once in a while ',' Love ',' Laughing '],
        [' everybody ', ' once in a while ', ' love ', ' crying '],
         [' it ', ' right ', ' love ', ' love ', ' love ']]

count_vec = CountVectorizer(tokenizer=lambda x: x, lowercase=False) # 한글 처리
m = count_vec.fit_transform(text)
```

Applications

```
#####################  TF-IDF: TfidfVectorizer #####################
from sklearn.feature_extraction.text import TfidfVectorizer
stop_words = stopwords.words('english')
len(stop_words)
stop_words

# Convert to document - word matrix via TF-IDF weights
vect = TfidfVectorizer ( stop_words = stop_words ). fit ( clean_review )
vect
x_train_vectorized = vect . transform ( clean_review )
x_train_vectorized.shape  # rows ( number of documents ) X columns ( number of words )
x_train_vectorized
print( x_train_vectorized )
vocab = vect.get_feature_names_out() # word names

import pandas as pd
df = pd.DataFrame ( x_train_vectorized.toarray ())
df
print( df )
```

Library function

❖ Introduction to expressing various word vectors using functions in the package

1. Tokenization function

2. Frequency vectors function (bag of words)
(1) Using NLTK
(2) Using
(3) Gensim Use

3. One - hot vector function
(1) Using
(2) Using
(3) Gensim Use

4. TF-IDF function
(1) Using NLTK
(2) Using
(3) Gensim Use

Library function

1. Tokenization function

```
import nltk
nltk.download('punkt')
import string

def tokenize(text):
     stem = nltk.stem.SnowballStemmer('english')   # stem extraction
    text = corpus[0].lower()  # into small letters

    for token in nltk.word_tokenize(text):
          if token in string.punctuation: continue  # string.punctuation = '!"#$%&'()*+,-./:;<=>?@[₩]^_`{|}~'
          yield stem.stem(token)

# The corpus object
corpus1 = [
     "The elephant sneezed at the sight of potatoes.",
     "Bats can see via echolocation. See the bat sight sneeze!",
     "Wondering, she opened the door to the studio.",
]

#########################
tokenize(corpus1[0])
list(tokenize(corpus1[0]))
print(list(tokenize(corpus1[0])))
```

Library function

2. Bag of words

(1) NLTK language

```
def nltk_frequency_vectorize (corpus):
# The NLTK frequency vectorize method
from collections import defaultdict
    def vectorize ( doc ):
features = defaultdict ( int )
for token in tokenize(doc):
features [ token ] += 1
        return features
    return map(vectorize, corpus)

print(list(nltk_frequency_vectorize(corpus1)))
```

(2) Sklearn

```
def sklearn_frequency_vectorize(corpus):
    # The Scikit-Learn frequency vectorize method
    from sklearn.feature_extraction.text import CountVectorizer

    vectorizer = CountVectorizer()
    return vectorizer.fit_transform(corpus)

matrix0 = sklearn_frequency_vectorize(corpus1)

import pandas as pd
df = pd.DataFrame(matrix0.toarray())
df
print(df)
```

Library function

(3) Gensim

```
def gensim_frequency_vectorize(corpus):
    # The Gensim frequency vectorize method
    import gensim

    tokenized_corpus = [list(tokenize(doc)) for doc in corpus]
    id2word = gensim.corpora.Dictionary(tokenized_corpus)
    return [id2word.doc2bow(doc) for doc in tokenized_corpus]

gensim_frequency_vectorize(corpus1)
list(gensim_frequency_vectorize(corpus1))
```

Library function

3. one-hot vector encoding

(1) NLTK

```
def nltk_one_hot_vectorize(corpus):
    def vectorize(doc):              # The NLTK one hot vectorize method
        return {
            token: True
            for token in tokenize(doc)
        }
    return map(vectorize, corpus)

result = nltk_one_hot_vectorize(corpus)
print(list(result))
```

(2) Sklearn

```
def sklearn_one_hot_vectorize(corpus):    # The Sklearn one hot vectorize method
    from sklearn.feature_extraction.text import CountVectorizer
    from sklearn.preprocessing import Binarizer
    freq    = CountVectorizer()
    vectors = freq.fit_transform(corpus)
    print(len(vectors.toarray()[0]))
    onehot  = Binarizer()
    vectors = onehot.fit_transform(vectors.toarray())

    print(len(vectors[0]))

sklearn_one_hot_vectorize(corpus1)
```

Library function

(3) Gensim

```
def gensim_one_hot_vectorize(corpus):
    # The Gensim one hot vectorize method
    import gensim
    import numpy as np

    corpus  = [list(tokenize(doc)) for doc in corpus]
    id2word = gensim.corpora.Dictionary(corpus)

    corpus  = np.array([
        [(token[0], 1) for token in id2word.doc2bow(doc)]
        for doc in corpus
    ])

    return corpus

gensim_one_hot_vectorize(corpus1)
```

Library function

4. TF-IDF

(1) NLTK

```
def nltk_tfidf_vectorize(corpus):
    from nltk.text import TextCollection
    corpus = [list(tokenize(doc)) for doc in corpus]
    texts = TextCollection(corpus)
    for doc in corpus:
        yield {
            term: texts.tf_idf(term, doc)
            for term in doc
        }

list(nltk_tfidf_vectorize(corpus1))
```

(2) Sklearn

```
def sklearn_tfidf_vectorize(corpus):
    from sklearn.feature_extraction.text import TfidfVectorizer

    tfidf = TfidfVectorizer()
    return tfidf.fit_transform(corpus)
###
matrix1 = sklearn_tfidf_vectorize(corpus1)

import pandas as pd
df = pd.DataFrame(matrix1.toarray())
df
print(df)
```