Wordpiece Model(WPM) 은 우리가 접한 적이 있는 아이디어를 기반으로 만들어졌습니다. 두 단어 `preview` 와 `predict` 를 보면 접두어인 `pre` 가 공통되고 있죠? `pre` 가 들어간 단어는 주로 **"미리", "이전의"** 와 연계되는 의미를 가지고 있습니다. 컴퓨터도 두 단어를 따로 볼 게 아니라 `pre+view` 와 `pre+dict` 로 본다면 학습을 더 잘 할 수 있지 않을까요?

이처럼 한 단어를 여러 개의 Subword의 집합으로 보는 방법이 WPM입니다. WPM의 원리를 알기 전, 먼저 알아야 할 것이 바로 **Byte Pair Encoding(BPE)** 입니다.

Byte Pair Encoding(BPE)

BPE 알고리즘이 고안된 것은 1994년입니다. 그때는 자연어 처리에 적용하기 위해서가 아니라 데이터 압축을 위해서 생겨났었죠. 데이터에서 **가장 많이 등장하는 바이트 쌍(Byte Pair)** 을 새로운 단어로 치환하여 압축하는 작업을 반복하는 방식으로 동작합니다. 예시는 아래와 같습니다.

```
aaabdaaabac # 가장 많이 등장한 바이트 쌍 "aa"를 "Z"로 치환합니다.
→
ZabdZabac   # "aa" 총 두 개가 치환되어 4바이트를 2바이트로 압축하였습니다.
Z=aa        # 그다음 많이 등장한 바이트 쌍 "ab"를 "Y"로 치환합니다.
→
ZYdZYac     # "ab" 총 두 개가 치환되어 4바이트를 2바이트로 압축하였습니다.
Z=aa        # 여기서 작업을 멈추어도 되지만, 치환된 바이트에 대해서도 진행한다면
Y=ab        # 가장 많이 등장한 바이트 쌍 "ZY"를 "X"로 치환합니다.
→
XdXac
Z=aa
Y=ab
X=ZY       # 압축이 완료되었습니다!
```

아주 직관적인 알고리즘이죠? 이를 토큰화에 적용하자고 제안한 것은 2015년이었습니다. 모든 단어를 문자(바이트)들의 집합으로 취급하여 자주 등장하는 문자 쌍을 합치면, 접두어나 접미어의 의미를 캐치할 수 있고, 처음 등장하는 단어는 문자(알파벳)들의 조합으로 나타내어 **OOV 문제를 완전히 해결**할 수 있다는 것이죠!

비교적 최근의 기술을 소개해드리는 만큼 논문을 함께 첨부합니다.

- Neural Machine Translation of Rare Words with Subword Units

위 논문은 Python 소스 코드를 함께 제공해 주어 간편하게 실습을 해 볼 수 있습니다. 논문에서 제공해 주는 예제로 동작 방식을 자세히 들여다보죠!

n-grams

- 유니그램 (1)
- 바이그램 (2)
- 트라이그램 (3)

나는 학교에 가서 공부를 합니다.

```
def n_grams(text, n):
    return [text[i:i+n] for i in range(len(text)-n+1)]

cleaned = ['mary', ',', "n't", 'slap', 'green', 'witch', '.']
print(n_grams(cleaned, 1))
```

```
[['mary'], [','], ["n't"], ['slap'], ['green'], ['witch'], ['.']]
```

BPE Algorithm

```
import re, collections
```

```
num_merges = 10 # BPE를 몇 회 수행할 것지 정함.
```

```
# BPE corpus
corpus = """
low lower newest widest
low lower newest widest
low       newest widest
low       newest
low       newest
          newest
"""
```

```
dictionary = {'l o w </w>' : 5,
              'l o w e r </w>' : 2,
              'n e w e s t </w>' : 6,
              'w i d e s t </w>' :3
}
```

```
def get_stats(dictionary):

    """
    단어 사전을 불러와
    단어는 공백 단위로 쪼개어 문자 list를 만들고
    빈도수와 쌍을 이루게 합니다. (symbols)
    """

  # 유니그램의 pair들의 빈도수를 카운트
    pairs = collections.defaultdict(int)
    for word, freq in dictionary.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
```

```
        for i in range(len(symbols)-1):
            pairs[symbols[i], symbols[i+1]] += freq
    print('현재 pair들의 빈도수 :', dict(pairs))
    return pairs
```

```
def merge_dictionary(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out
```

```
bpe_codes = {}
bpe_codes_reverse = {}
for i in range(num_merges):
    print(">> Step {0}".format(i+1))
    pairs = get_stats(dictionary)
    best = max(pairs, key=pairs.get)
    dictionary = merge_dictionary(best, dictionary)

    bpe_codes[best] = i
    bpe_codes_reverse[best[0] + best[1]] = best

    print("new merge: {}".format(best))
    print("dictionary: {}".format(dictionary))
```

```
>> Step 1
현재 pair들의 빈도수 : {('l', 'o'): 7, ('o', 'w'): 7, ('w', '</w>'): 5, ('w', 'e')
new merge: ('e', 's')
dictionary: {'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w es t </w>': 6, 'w i
>> Step 2
현재 pair들의 빈도수 : {('l', 'o'): 7, ('o', 'w'): 7, ('w', '</w>'): 5, ('w', 'e')
new merge: ('es', 't')
dictionary: {'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w est </w>': 6, 'w i d
>> Step 3
현재 pair들의 빈도수 : {('l', 'o'): 7, ('o', 'w'): 7, ('w', '</w>'): 5, ('w', 'e')
new merge: ('est', '</w>')
dictionary: {'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w est</w>': 6, 'w i d
>> Step 4
현재 pair들의 빈도수 : {('l', 'o'): 7, ('o', 'w'): 7, ('w', '</w>'): 5, ('w', 'e')
new merge: ('l', 'o')
dictionary: {'lo w </w>': 5, 'lo w e r </w>': 2, 'n e w est</w>': 6, 'w i d es
>> Step 5
현재 pair들의 빈도수 : {('lo', 'w'): 7, ('w', '</w>'): 5, ('w', 'e'): 2, ('e', 'r'
new merge: ('lo', 'w')
dictionary: {'low </w>': 5, 'low e r </w>': 2, 'n e w est</w>': 6, 'w i d est<
>> Step 6
현재 pair들의 빈도수 : {('low', '</w>'): 5, ('low', 'e'): 2, ('e', 'r'): 2, ('r',
new merge: ('n', 'e')
dictionary: {'low </w>': 5, 'low e r </w>': 2, 'ne w est</w>': 6, 'w i d est</
>> Step 7
현재 pair들의 빈도수 : {('low', '</w>'): 5, ('low', 'e'): 2, ('e', 'r'): 2, ('r',
new merge: ('ne', 'w')
dictionary: {'low </w>': 5, 'low e r </w>': 2, 'new est</w>': 6, 'w i d est</w
```

```
>> Step 8
현재 pair들의 빈도수 : {('low', '</w>'): 5, ('low', 'e'): 2, ('e', 'r'): 2, ('r',
new merge: ('new', 'est</w>')
dictionary: {'low </w>': 5, 'low e r </w>': 2, 'newest</w>': 6, 'w i d est</w>
>> Step 9
현재 pair들의 빈도수 : {('low', '</w>'): 5, ('low', 'e'): 2, ('e', 'r'): 2, ('r',
new merge: ('low', '</w>')
dictionary: {'low</w>': 5, 'low e r </w>': 2, 'newest</w>': 6, 'w i d est</w>'
>> Step 10
현재 pair들의 빈도수 : {('low', 'e'): 2, ('e', 'r'): 2, ('r', '</w>'): 2, ('w', 'i
new merge: ('w', 'i')
dictionary: {'low</w>': 5, 'low e r </w>': 2, 'newest</w>': 6, 'wi d est</w>':
```

```
print(bpe_codes)
```

```
{('e', 's'): 0, ('es', 't'): 1, ('est', '</w>'): 2, ('l', 'o'): 3, ('lo', 'w')
```

OOV 대처하기

만일 `lowest` 라는 처음 보는 단어가 등장하더라도, 위 알고리즘을 따르면 어느 정도 의미가 파악된 `low` 와 `est` 의 결합으로 표현할 수 있습니다. 또 BPE의 놀라운 점은 아무리 큰 데이터도 원하는 크기로 OOV 문제없이 사전을 정의할 수 있다는 것입니다. 극단적으로 생각했을 때 알파벳 26개와 특수문자, 문장부호를 아무리 추가해도 100개 이내로 사전을 정의할 수 있죠.(물론 그러면 안 됩니다!!)

Embedding 레이어는 **단어의 개수 x Embedding 차원 수** 의 Weight를 생성하기 때문에 단어의 개수가 줄어드는 것은 곧 메모리의 절약으로 이어집니다. 많은 데이터가 곧 정확도로 이어지기 때문에 이런 기여는 굉장히 의미가 있습니다!

하지만 아직도! 완벽하다고는 할 수 없습니다. 만약 수많은 데이터를 사용해 만든 BPE 사전으로 모델을 학습시키고 문장을 생성하게 했다고 합시다. 그게 `[i, am, a, b, o, y, a, n, d, you, are, a, gir, l]` 이라면, 어떤 기준으로 이들을 결합해서 문장을 복원하죠? 몽땅 한꺼번에 합쳤다간 끔찍한 일이 벌어질 것만 같습니다...

```
def get_pairs(word):
  pairs = set()
  prev_char = word[0]
  for char in word[1:]:
    pairs.add((prev_char, char))
    prev_char = char
  return pairs
```

```
orig = 'hi'
word = tuple(orig) + ('</w>',)
print(word)
```

```
('h', 'i', '</w>')
```

```
def encode(orig):
  word = tuple(orig) + ('</w>',)
```

```
  word = tuple(orig) + ('</w>',)
  print("__word split into characters:__ <tt>{}<tt>".format(word))

  pairs = get_pairs(word)

  if not pairs:
    return orig

  iteration = 0
  while True:
    iteration += 1
    print("__Iteration {}:__".format(iteration))

    print("Bigram in the word: {}".format(pairs))
    bigram = min(pairs, key = lambda pair: bpe_codes.get(pair, float('inf')))
    print("candidate for merging: {}".format(bigram))
    if bigram not in bpe_codes:
      print("__Candidate not in BPE merges, algorithm stops.__")
      break
    first, second = bigram
    new_word = []
    i = 0
    while i < len(word):
      try:
        j = word.index(first, i)
        new_word.extend(word[i:j])
        i = j
      except:
        new_word.extend(word[i:])
        break

      if word[i] == first and i < len(word)-1 and word[i+1] == second:
        new_word.append(first+second)
        i += 2
      else:
        new_word.append(word[i])
        i += 1
    new_word = tuple(new_word)
    word = new_word
    print("word after merging : {}".format(word))
    if len(word) == 1:
      break
    else:
      pairs = get_pairs(word)

  # 특별토큰인 </w>는 출력하지 않는다.
  if word[-1] == '</w>':
    word = word[:-1]
  elif word[-1].endswith('</w>'):
    word = word[:-1] + (word[-1].replace('</w>', ''), )
  return word
```

단어 'loki'가 들어오면 BPE 알고리즘 해당 단어를 어떻게 분리할까요?

```
1 encode("loki")
```

```
__word split into characters:__ <tt>('l', 'o', 'k', 'i', '</w>')<tt>
__Iteration 1:__
Bigram in the word: {('o', 'k'), ('l', 'o'), ('i', '</w>'), ('k', 'i')}
candidate for merging: ('l', 'o')
word after merging : ('lo', 'k', 'i', '</w>')
__Iteration 2:__
Bigram in the word: {('i', '</w>'), ('lo', 'k'), ('k', 'i')}
candidate for merging: ('i', '</w>')
__Candidate not in BPE merges, algorithm stops.__
('lo', 'k', 'i')
```

현재 서브워드 단어집합에는 'lo'가 존재하므로, 'lo'는 유지하고 'k'와 'i'는 분리시킵니다. 단어 'lowest'에 대해서도 수행해봅시다.

```
1 encode("lowest")
```

```
__word split into characters:__ <tt>('l', 'o', 'w', 'e', 's', 't', '</w>')<tt>
__Iteration 1:__
Bigram in the word: {('t', '</w>'), ('w', 'e'), ('l', 'o'), ('e', 's'), ('s',
candidate for merging: ('e', 's')
word after merging : ('l', 'o', 'w', 'es', 't', '</w>')
__Iteration 2:__
Bigram in the word: {('t', '</w>'), ('l', 'o'), ('w', 'es'), ('es', 't'), ('o'
candidate for merging: ('es', 't')
word after merging : ('l', 'o', 'w', 'est', '</w>')
__Iteration 3:__
Bigram in the word: {('l', 'o'), ('est', '</w>'), ('w', 'est'), ('o', 'w')}
candidate for merging: ('est', '</w>')
word after merging : ('l', 'o', 'w', 'est</w>')
__Iteration 4:__
Bigram in the word: {('l', 'o'), ('o', 'w'), ('w', 'est</w>')}
candidate for merging: ('l', 'o')
word after merging : ('lo', 'w', 'est</w>')
__Iteration 5:__
Bigram in the word: {('w', 'est</w>'), ('lo', 'w')}
candidate for merging: ('lo', 'w')
word after merging : ('low', 'est</w>')
__Iteration 6:__
Bigram in the word: {('low', 'est</w>')}
candidate for merging: ('low', 'est</w>')
__Candidate not in BPE merges, algorithm stops.__
('low', 'est')
```

현재 서브워드 단어집합에 'low'와 'est'가 존재하므로, 'low'와 'est'를 분리시킵니다. 단어 'lowing'에 대해서도 수행해봅시다.

```
1 encode("lowing")
```

```
__word split into characters:__ <tt>('l', 'o', 'w', 'i', 'n', 'g', '</w>')<tt>
__Iteration 1:__
Bigram in the word: {('i', 'n'), ('l', 'o'), ('w', 'i'), ('g', '</w>'), ('o',
candidate for merging: ('l', 'o')
```

```
word after merging : ('lo', 'w', 'i', 'n', 'g', '</w>')
__Iteration 2:__
Bigram in the word: {('i', 'n'), ('w', 'i'), ('g', '</w>'), ('lo', 'w'), ('n',
candidate for merging: ('lo', 'w')
word after merging : ('low', 'i', 'n', 'g', '</w>')
__Iteration 3:__
Bigram in the word: {('n', 'g'), ('g', '</w>'), ('i', 'n'), ('low', 'i')}
candidate for merging: ('n', 'g')
__Candidate not in BPE merges, algorithm stops.__
('low', 'i', 'n', 'g')
```

현재 서브워드 단어집합에 'low'가 존재하지만, 'i', 'n', 'g'의 바이그램 조합으로 이루어진 서브워드는 존재하지 않으므로 'i', 'n', 'g'로 전부 분리합니다. 훈련된 데이터 중에서 어떤 서브워드도 존재하지 않는 'highing'은 어떨까요?

```
1 encode("highing")
```

```
__word split into characters:__ <tt>('h', 'i', 'g', 'h', 'i', 'n', 'g', '</w>'
__Iteration 1:__
Bigram in the word: {('i', 'n'), ('h', 'i'), ('g', '</w>'), ('i', 'g'), ('n',
candidate for merging: ('i', 'n')
__Candidate not in BPE merges, algorithm stops.__
('h', 'i', 'g', 'h', 'i', 'n', 'g')
```

Wordpiece Model(WPM)

이에 구글에서 BPE를 변형해 제안한 알고리즘이 바로 WPM입니다. WPM은 BPE에 대해 **두 가지 차별성**을 가집니다.

1. 공백 복원을 위해 단어의 시작 부분에 언더바 _ 를 추가합니다.
2. 빈도수 기반이 아닌 가능도(Likelihood)를 증가시키는 방향으로 문자 쌍을 합칩니다.

첫 번째 문항은 아주 쉬운 내용으로, 앞서 사용한 예문을 빌리면 `[_i, _am, _a, _b, o, y, _a, n, d, _you, _are, _a, _gir, l]`로 토큰화를 한다는 것입니다. 이렇게 하면 문장을 복원하는 과정이 **1) 모든 토큰을 합친 후, 2) 언더바 _를 공백으로 치환**으로 마무리되어 간편하죠.

두 번째 문항은 다소 난해하게 다가올 수도 있습니다. 직관적인 이해를 얻고 넘어가는 것을 목표로 하죠. 본 내용은 아래 논문 3절과 4절에 자세하게 나와 있습니다.

- JAPANESE AND KOREAN VOICE SEARCH

(여기서 잠깐) 구글에서 이 기법을 한국어, 일본어 텍스트 처리를 위해 고려했다는 사실이 흥미롭지 않나요? 이 점은 2가지를 시사합니다.

- 조사, 어미 등의 활용이 많고 복잡한 한국어 같은 모델의 토크나이저로 WPM이 좋은 대안이 될 수 있다.
- WPM은 어떤 언어든 무관하게 적용 가능한 language-neutral하고 general한 기법이다. 한국어 형태소 분석기처럼 한국어에만 적용 가능한 기법보다 훨씬 활용도가 크다.

WPM을 수행하기 이전의 문장: Jet makers feud over seat width with big orders at stake
WPM을 수행한 결과(wordpieces): _J et _makers _fe ud _over _seat _width _with _big _orders _at _stake

Jet는 J와 et로 나누어졌으며, feud는 fe와 ud로 나누어진 것을 볼 수 있습니다. WPM은 모든 단어의 맨 앞에 _를 붙이고, 단어는 서브 워드(subword)로 통계에 기반하여 띄어쓰기로 분리합니다. 여기서 언더바 _는 문장 복원을 위한 장치입니다.

예컨대, WPM의 결과로 나온 문장을 보면, Jet → _J et와 같이 기존에 없던 띄어쓰기가 추가되어 서브 워드(subwords)들을 구분하는 구분자 역할을 하고 있습니다. 그렇다면 기존에 있던 띄어쓰기와 구분자 역할의 띄어쓰기는 어떻게 구별할까요? 이 역할을 수행하는 것이 단어들 앞에 붙은 언더바 _입니다. WPM이 수행된 결과로부터 다시 수행 전의 결과로 돌리는 방법은 현재 있는 모든 띄어쓰기를 전부 제거하고, 언더바를 띄어쓰기로 바꾸면 됩니다.

이 알고리즘은 최신 딥 러닝 모델 BERT를 훈련하기 위해서 사용되기도 하였습니다.

토큰화의 끝판왕으로 보이는 이 WPM은 아쉽게도 공개되어 있지는 않습니다. 대신에 구글의 *SentencePiece* 라이브러리를 통해 고성능의 BPE를 사용할 수 있습니다! *SentencePiece*에는 전처리 과정도 포함되어 있어서, 데이터를 따로 정제할 필요가 없어 간편하기까지 합니다. 따라서 아래 깃허브 페이지에 방문해 사용법을 간단히 눈으로 봐두도록 하죠! 아마 다음 프로젝트 노드에서는 *SentencePiece* 라이브러리를 적극적으로 활용하는 실습을 진행하게 될 것입니다.

- google/sentencepiece

이제 우리는 어떤 언어에도 OOV 발생 우려 없이 안정적으로 활용할 수 있는 멋진 토크나이징 기술을 확보했습니다. 이제는 컴퓨터가 단어사전을 안심하고 활용할 수 있겠군요!

하지만 아직 우리는 제대로 된 단어의 분산 표현을 얻는 법을 제대로 다루지는 않았습니다. 이쯤에서 이런 고민이 생기게 됩니다. 한국어라면 `자동차`를 `_자동 / 차` 로 분리되는데... 속성이 아무리 추상적이래도 보기에 `차`가 마시는 차인지, 달리는 차인지 도통 알 수가 없죠? 게다가 설령 토큰화가 완벽하다고 해도, `남자`가 `[-1, 0]`인지 `[1, 0]`인지는 컴퓨터 입장에서는 알 도리가 없습니다.

Embedding 레이어는 선언 즉시 랜덤한 실수로 Weight 값을 채우고, 학습을 진행하며 적당히 튜닝해가는 방식으로 속성을 맞춰가지만 이는 뭔가 찜찜합니다. 토큰들이 멋지게 의미를 갖게 하는 방법은 없을까요?

IMDB리뷰 tensorflow의 subwordTextEncoder로 토큰화

[ ] *숨겨진 셀 17개*

네이버 영화 리뷰로 서브워드토크나이저

[ ] *숨겨진 셀 13개*

IMDB리뷰 sentencePiece로 토큰화

```
1 !pip install sentencepiece
```

```
Collecting sentencepiece
```

```
  Downloading sentencepiece-0.1.96-cp37-cp37m-manylinux_2_17_x86_64.manylinux2
     |████████████████████████████████████████| 1.2 MB 33.8 MB/s
Installing collected packages: sentencepiece
Successfully installed sentencepiece-0.1.96
```

```
1 import sentencepiece as spm
2 import pandas as pd
3 import urllib.request
4 import csv
```

```
1 urllib.request.urlretrieve("https://raw.githubusercontent.com/LawrenceDuan/IMDb-
```

```
('IMDb_Reviews.csv', <http.client.HTTPMessage at 0x7fea90b77d50>)
```

```
1 train_df = pd.read_csv('IMDb_Reviews.csv')
```

```
1 train_df.head()
```

	review	sentiment
0	My family and I normally do not watch local mo...	1
1	Believe it or not, this was at one time the wo...	0
2	After some internet surfing, I found the "Home...	0
3	One of the most unheralded great works of anim...	1
4	It was the Sixties, and anyone with long hair ...	0

```
1 print('리뷰 개수 : ', len(train_df))
```

```
리뷰 개수 :  50000
```

```
1 with open('imdb_review.txt', 'w', encoding='utf8') as f:
2   f.write('\n'.join(train_df['review']))
```

sentencepiece의 자세한 실행 옵션 : https://github.com/google/sentencepiece

```
spm.SentencePieceTrainer.train(
    f"--input={corpus} --model_prefix={prefix} --vocab_size={vocab_size + 7}" +
    " --model_type=bpe" +
    " --max_sentence_length=999999" + # 문장 최대 길이
    " --pad_id=0 --pad_piece=[PAD]" + # pad (0)
    " --unk_id=1 --unk_piece=[UNK]" + # unknown (1)
    " --bos_id=2 --bos_piece=[BOS]" + # begin of sequence (2)
    " --eos_id=3 --eos_piece=[EOS]" + # end of sequence (3)
    " --user_defined_symbols=[SEP],[CLS],[MASK]") # 사용자 정의 토큰
```

```
spm.SentencePieceTrainer.Train('--input=imdb_review.txt --model_prefix=imdb --vocab_size=5
```

```
corpus = "imdb_review.txt" #입력 corpus
prefix = "imdb" # 저장할 단어장 이름
vocab_size = 5000
spm.SentencePieceTrainer.Train(
    f"--input={corpus} --model_prefix={prefix} --vocab_size={vocab_size}" +
    " --model_type=bpe" +
    " --max_sentence_length=9999") # 문장 최대 길이
```

- input : 학습시킬 파일
- model_prefix : 만들어질 모델 이름
- vocab_size : 단어집합크기
- model_type : 사용할 모델 (unigram(default), bpe, char, word)
- pad_id, pad_piece : pad token id, 값
- unk_id, unk_piece : unknown token id, 값
- bos_id, bos_piece : begin of sentence token id,값
- eos_id, eos_piece : end of sequence token id, 값
- user_defined_symbols : 사용자 정의 토큰

```
vocab_list = pd.read_csv('imdb.vocab', sep='\t', header=None, quoting=csv.QUOTE_
vocab_list.sample(10)
```

	0	1
2997	__tort	-2994
1498	__brilliant	-1495
471	__being	-468
2511	__public	-2508
4650	__theatre	-4647
3223	__exception	-3220
4581	John	-4578
3664	aint	-3661
4260	__brutal	-4257
4589	__hadn	-4586

```
len(vocab_list)
```

```
sp = spm.SentencePieceProcessor()
vocab_file = "imdb.model"
sp.load(vocab_file)
```

```
sp.load(vocab_file)
```

```
True
```

```
lines = [
        "I didn't at all think of it this way.",
        "I have waited a long time for someone to film"
]
for line in lines:
  print(line)
  print(sp.encode_as_pieces(line)) # 문장을 입력하면 서브워드 시퀀스로 변환
  print(sp.encode_as_ids(line)) # 문장을 입력하면 정수 시퀀스로 변환
  print()
```

```
I didn't at all think of it this way.
['_I', '_didn', "'", 't', '_at', '_all', '_think', '_of', '_it', '_this', '_wa
[41, 623, 4950, 4926, 138, 169, 378, 30, 58, 73, 413, 4945]

I have waited a long time for someone to film
['_I', '_have', '_wa', 'ited', '_a', '_long', '_time', '_for', '_someone', '_t
[41, 141, 1364, 1120, 4, 666, 285, 92, 1078, 33, 91]
```

```
sp.GetPieceSize() #단어집합의 크기
```

```
5000
```

```
sp.IdToPiece(120) #정수로부터 매핑되는 서브워드 변환
```

```
'_not'
```

```
# 위에서 출력된 결과를 sp.PieceTold 안으로 입력하기
sp.PieceToId('_not') # 서브워드로부터 매핑되는 정수로 변환
```

```
120
```

```
# 위에서 출력된 정수 시퀀스를 sp.DecodeIds의 정수 시퀀스에 입력하기
sp.DecodeIds([41, 141, 1364, 1120, 4, 666, 285, 92, 1078, 33, 91])
```

```
'I have waited a long time for someone to film'
```

```
# 서브워드 시퀀스로부터 문장으로 변환
sp.DecodePieces(['_I', '_have', '_wa', 'ited', '_a', '_long', '_time', '_for', '
```

```
'I have waited a long time for someone to film'
```

```
print(sp.encode('I have waited a long time for someone to film', out_type=str))
print(sp.encode('I have waited a long time for someone to film', out_type=int))
```

```
['_I', '_have', '_wa', 'ited', '_a', '_long', '_time', '_for', '_someone', '_t
[41, 141, 1364, 1120, 4, 666, 285, 92, 1078, 33, 91]
```

네이버 영화리뷰로 센텐스피스 적용하기

```
1 import pandas as pd
2 import sentencepiece as spm
3 import urllib.request
4 import csv
```

```
1 urllib.request.urlretrieve("https://raw.githubusercontent.com/e9t/nsmc/master/ra
```

```
('ratings.txt', <http.client.HTTPMessage at 0x7fea92634410>)
```

```
1 naver_df = pd.read_table('ratings.txt')
2 naver_df.head()
```

	id	document	label
0	8112052	어릴때보고 지금다시봐도 재밌어요ㅋㅋ	1
1	8132799	디자인을 배우는 학생으로, 외국디자이너와 그들이 일군 전통을 통해 발전해가는 문화산...	1
2	4655635	폴리스스토리 시리즈는 1부터 뉴까지 버릴께 하나도 없음.. 최고.	1
3	9251303	와.. 연기가 진짜 개쩔구나.. 지루할거라고 생각했는데 몰입해서 봤다.. 그래 이런...	1
4	10067386	안개 자욱한 밤하늘에 떠 있는 초승달 같은 영화.	1

```
1 # 리뷰 갯수 출력하기
2 print('리뷰 개수 :',len(naver_df)) # 리뷰 개수 출력
```

```
리뷰 개수 : 200000
```

```
1 # Null 존재하는지 확인하기
2 print(naver_df.isnull().values.any())
```

```
True
```

```
1 # Null값이 존재하는 행 제거
2 naver_df = naver_df.dropna(how = 'any')
3 # Null 존재하는지 확인하기
4 print(naver_df.isnull().values.any())
```

```
False
```

```
1 # 리뷰 갯수 출력하기
2 print('리뷰 개수 :',len(naver_df))
```

```
리뷰 개수 : 199992
```

```
1 with open('naver_review.txt', 'w', encoding='utf8') as f:
2     f.write('\n'.join(naver_df['document']))
```

```
spm.SentencePieceTrainer.Train('--input=naver_review.txt --model_prefix=naver --
```

```
# vocab불러오기 pd.read_csv
vocab_list = pd.read_csv('naver.vocab', sep='\t', header=None, quoting=csv.QUOTE
#vocab_list[:10]
vocab_list.head(10)
# vocab 10개출력
```

	0	1
0	<unk>	0
1	<s>	0
2	</s>	0
3	..	0
4	영화	-1
5	▁영화	-2
6	▁이	-3
7	▁아	-4
8	...	-5
9	▁그	-6

```
sp = spm.SentencePieceProcessor()
vocab_file = "naver.model"
sp.load(vocab_file)
```

```
True
```

```
lines = [
  "뭐 이딴 것도 영화냐.",
  "진짜 최고의 영화입니다 ㅋㅋ",
]
for line in lines:
  print(line)
  print(sp.encode_as_pieces(line))
  print(sp.encode_as_ids(line))
  print()
```

```
뭐 이딴 것도 영화냐.
['▁뭐', '▁이딴', '▁것도', '▁영화냐', '.']
[132, 966, 1296, 2590, 3276]

진짜 최고의 영화입니다 ㅋㅋ
['▁진짜', '▁최고의', '▁영화입니다', '▁ㅋㅋ']
[54, 200, 821, 85]
```

```
1 sp.GetPieceSize()
```

```
5000
```

```
1 sp.IdToPiece(4)
```

```
'영화'
```

```
1 sp.PieceToId('영화')
```

```
4
```

```
1 sp.DecodeIds([132, 966, 1296, 2590, 3276])
```

```
'뭐 이딴 것도 영화냐.'
```

```
1 sp.DecodePieces(['▁뭐', '▁이딴', '▁것도', '▁영화냐', '.']) # 서브워드 시퀀스 입력
```

```
'뭐 이딴 것도 영화냐.'
```

```
1 print(sp.encode('진짜 최고의 영화입니다 ㅋㅋ', out_type=str))
2 print(sp.encode('진짜 최고의 영화입니다 ㅋㅋ', out_type=int))
```

```
['▁진짜', '▁최고의', '▁영화입니다', '▁ㅋㅋ']
[54, 200, 821, 85]
```

one-hot encoding실습

```
1 import numpy as np
```

```
1 # 입력 문장
2 raw_inputs = [
3               "나는 학생 입니다",
4               "나는 좋은 선생님 입니다",
5               "당신은 매우 좋은 선생님 입니다"
6 ]
7
8 # 정답 학생(1) 기타(0)
9 raw_labels = [1, 0, 0]
```

```
1 words = []
2 for s in raw_inputs:
3     words.extend(s.split())
4
5 # 중복 단어 제거
6 words = list(dict.fromkeys(words))
7
8 word_to_id = {"[PAD]":0, "[UNK]":1}
```

```
for w in words:
    word_to_id[w] = len(word_to_id)

# 각 번호별 단어
id_to_word = {i: w for w, i in word_to_id.items()}

print(id_to_word)
```

```
{0: '[PAD]', 1: '[UNK]', 2: '나는', 3: '학생', 4: '입니다', 5: '좋은', 6: '선생님',
```

```
train_inputs = []
for s in raw_inputs:
    row = [word_to_id[w] for w in s.split()]
    row += [0] * (5- len(row))
    train_inputs.append(row)

train_inputs = np.array(train_inputs)

print(train_inputs)
```

```
[[2 3 4 0 0]
 [2 5 6 4 0]
 [7 8 5 6 4]]
```

```
# one -hot metrix 생성
onehot_metrix = np.eye(len(word_to_id))
print(onehot_metrix)
```

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

```
train_onehots = onehot_metrix[train_inputs]
print(train_onehots)
```

```
[[[0. 0. 1. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 1. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0. 0. 0.]
  [1. 0. 0. 0. 0. 0. 0. 0. 0.]
  [1. 0. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 1. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0. 0. 0.]
  [1. 0. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0. 0. 0. 1. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

```
       [0. 0. 0. 0. 0. 1. 0. 0. 0.]
       [0. 0. 0. 0. 0. 0. 1. 0. 0.]
       [0. 0. 0. 0. 1. 0. 0. 0. 0.]]]
```

```
1 print(np.argmax(train_onehots, axis=-1))
```

```
   [[2 3 4 0 0]
    [2 5 6 4 0]
    [7 8 5 6 4]]
```

```
1 x = np.argmax(train_onehots, axis=-1)
```

```
1 import tensorflow as tf
2 import tensorflow.keras.layers as L
```

```
1 x_len = train_onehots.shape
```

```
1 inp = tf.convert_to_tensor(x, dtype=tf.int32)
2 inp_len = tf.convert_to_tensor(x_len, dtype=tf.int32)
```

```
1 inp, inp_len
```

```
   (<tf.Tensor: shape=(3, 5), dtype=int32, numpy=
    array([[2, 3, 4, 0, 0],
           [2, 5, 6, 4, 0],
           [7, 8, 5, 6, 4]], dtype=int32)>,
    <tf.Tensor: shape=(3,), dtype=int32, numpy=array([3, 5, 9], dtype=int32)>)
```

```
1 vocab = 1000
2 dim = 3
3 embed = L.Embedding(vocab, dim)
```

```
1 embed(inp)
```

```
   <tf.Tensor: shape=(3, 5, 3), dtype=float32, numpy=
   array([[[ 0.0399647 ,  0.0347276 ,  0.03682219],
           [-0.00117704, -0.02982651, -0.02315493],
           [-0.01293083,  0.04588192, -0.0193073 ],
           [-0.0154045 ,  0.00741573,  0.02601862],
           [-0.0154045 ,  0.00741573,  0.02601862]],

          [[ 0.0399647 ,  0.0347276 ,  0.03682219],
           [-0.01873317,  0.03784363, -0.00091445],
           [ 0.03667052, -0.00820971,  0.02016015],
           [-0.01293083,  0.04588192, -0.0193073 ],
           [-0.0154045 ,  0.00741573,  0.02601862]],

          [[ 0.03118033,  0.04501085,  0.02484627],
           [-0.006697  ,  0.03062637,  0.03237622],
           [-0.01873317,  0.03784363, -0.00091445],
           [ 0.03667052, -0.00820971,  0.02016015],
           [-0.01293083,  0.04588192, -0.0193073 ]]], dtype=float32)>
```

Bag of Words (Bow)

단어의 등장 순서를 고려하지 않는 빈도수 기반의 단어 표현 방법

1. 각 단어의 고유한 정수 인덱스를 부여
2. 각 인덱스 위치에 단어 토큰의 등장 횟수를 기록한 벡터를 만든다.

doc1 = 'John likes to watch movies. Mary likes movies too'

Bow1 = {"John" : 1, "likes : 2, "to" : 1, "watch": 1, "movies" 2, "Mary : 1, "too":1}

```
1 !pip install konlpy
```

```
Collecting konlpy
  Downloading konlpy-0.5.2-py2.py3-none-any.whl (19.4 MB)
     |████████████████████████████████| 19.4 MB 1.3 MB/s
Collecting beautifulsoup4==4.6.0
  Downloading beautifulsoup4-4.6.0-py3-none-any.whl (86 kB)
     |████████████████████████████████| 86 kB 5.7 MB/s
Requirement already satisfied: numpy>=1.6 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: lxml>=4.1.0 in /usr/local/lib/python3.7/dist-pa
Collecting colorama
  Downloading colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting JPype1>=0.7.0
  Downloading JPype1-1.3.0-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.w
     |████████████████████████████████| 448 kB 63.1 MB/s
Requirement already satisfied: tweepy>=3.7.0 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/d
Requirement already satisfied: requests[socks]>=2.11.1 in /usr/local/lib/pytho
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/pyth
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in /usr/local/lib/python
Installing collected packages: JPype1, colorama, beautifulsoup4, konlpy
  Attempting uninstall: beautifulsoup4
    Found existing installation: beautifulsoup4 4.6.3
    Uninstalling beautifulsoup4-4.6.3:
      Successfully uninstalled beautifulsoup4-4.6.3
Successfully installed JPype1-1.3.0 beautifulsoup4-4.6.0 colorama-0.4.4 konlpy
```

```
1 from konlpy.tag import Okt
2 import re
3 okt = Okt()
4
5 # 정규표현식을 통해 온점을 제거하는 정제 작업
6 token = re.sub("(\.)", "", "소비자는 주로 소비하는 상품을 기준으로 물가상승률을 느낀다.")
7
8 token = okt.morphs(token)
```

```
token = okt.morphs(token)

word2index = {}
bow = []

for voca in token:
    if voca not in word2index.keys():
        word2index[voca] = len(word2index)
        # token을 읽으면서, word2index에 없는 단어는 새로 추가하고, 이미 있는 단어는 넘긴다.
        bow.insert(len(word2index)-1, 1)
        # bow전체에 전부 기본값 1을 넣어준다. 단어갯수는 최소 1개이상이기 때문에
    else:
        index = word2index.get(voca)
        # 재 등장하는 단어의 인덱스를 받아오기
        bow[index] = bow[index]+1
        # 재 등장하는 단어는 해당하는 인덱스의 위치에 1을 더해줌 (단어갯수를 세는 것)

print(word2index)
```

```
{'소비자': 0, '는': 1, '주로': 2, '소비': 3, '하는': 4, '상품': 5, '을': 6, '기준': 7,
```

```
bow
```

```
[1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1]
```

tensorflow의 keras Tokenizer를 활용한 BOW

```
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
sentence = ["John likes to watch movies. \
            Mary likes movies too! \
            Mary also likes to watch football games."]
```

```
def print_bow(sentence):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(sentence) #단어장 생성
    bow = dict(tokenizer.word_counts) # 각 단어와 각 단어의 빈도를 bow를 저장

    print("Bag of words : ", bow)
    print("단어장(vocabulary)의 크기 :", len(tokenizer.word_counts)) #중복을 제거한 단어
```

```
print_bow(sentence)
```

```
Bag of words :  {'john': 1, 'likes': 3, 'to': 2, 'watch': 2, 'movies': 2, 'mar
단어장(vocabulary)의 크기 : 10
```

scikit-learn의 CountVectorizer을 활용한 BoW

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
sentence = ["John likes to watch movies. Mary likes movies too! Mary also likes

vector = CountVectorizer()

print("Bagg of Words : ", vector.fit_transform(sentence).toarray()) #코퍼스로부터 각
print("각 단어의 인덱스 :", vector.vocabulary_) #각 단어의 인덱스가 어떻게 부여되는지 보여줌
```

```
Bagg of Words :  [[1 1 1 1 3 2 2 2 1 2]]
각 단어의 인덱스 : {'john': 3, 'likes': 4, 'to': 7, 'watch': 9, 'movies': 6, 'mary
```

불용어를 제거한 BOW 만들기

사용자가 직접 정의한 불용어 사용

```
from sklearn.feature_extraction.text import CountVectorizer

text = ["Family is not an important thing. It's everything."]

vect = CountVectorizer(stop_words=["the", "a", "an", "is", "not"])
print(vect.fit_transform(text).toarray())
print(vect.vocabulary_)
```

```
[[1 1 1 1 1]]
{'family': 1, 'important': 2, 'thing': 4, 'it': 3, 'everything': 0}
```

CountVectorizer에서 제공하는 자체 불용어 사용

```
from sklearn.feature_extraction.text import CountVectorizer

text = ["Family is not an important thing. It's everything."]

vect = CountVectorizer(stop_words="english")
print(vect.fit_transform(text).toarray())
print(vect.vocabulary_)
```

```
[[1 1 1]]
{'family': 0, 'important': 1, 'thing': 2}
```

NLTK에서 지원하는 불용어 사용

```
!pip install nltk
import nltk
nltk.download('stopwords')
```

```
Requirement already satisfied: nltk in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (
[nltk_data] Downloading package stopwords to /root/nltk_data...
```

```
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

```
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import stopwords

text = ["Family is not an important thing. It's everything."]

sw = stopwords.words("english")
vect = CountVectorizer(stop_words= sw)
print(vect.fit_transform(text).toarray())
print(vect.vocabulary_)
```

```
[[1 1 1 1]]
{'family': 1, 'important': 2, 'thing': 3, 'everything': 0}
```

DTM (Document-Term Matrix)

다수의 문서에서 등장하는 각 단어들의 빈도를 행렬로 표현한 것
다수의 문서에 대해서 Bow를 하나의 행렬로 표현하고 부르는 용어

문서 1 : I like dog
문서 2 : I like cat
문서 3 : I like cat I like cat

```
import pandas as pd
content = [[0, 1, 1, 1], [1, 0, 1, 1], [2, 0, 2, 2,]]
df = pd.DataFrame(content)
df.index = ['(문서1) I like dog', '(문서2) I like cat', '(문서3) I like cat I like c
df.columns = ['cat', 'dog', 'I', 'like']
df
```

	cat	dog	I	like
(문서1) I like dog	0	1	1	1
(문서2) I like cat	1	0	1	1
(문서3) I like cat I like cat	2	0	2	2

```
import numpy as np
from numpy import dot
from numpy.linalg import norm

doc1 = np.array([0, 1, 1, 1])
doc2 = np.array([1, 0, 1, 1])
doc3 = np.array([2, 0, 2, 2])

def cos_sim(A, B): # 코사인 유사도는 0~1사이의 값을 가지고, 1에 가까울수록 유사도 높다고 판단
```

```
    return dot(A,B)/(norm(A) * norm(B))
```

```
print(cos_sim(doc1, doc2))
print(cos_sim(doc1, doc3))
print(cos_sim(doc2, doc3))
```

```
0.6666666666666667
0.6666666666666667
1.0000000000000002
```

scikit-learn CountVectorizer를 활용한 DTM구현

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
          'John likes to watch movies',
          'Mary likes movies too',
          'Mary also likes to watch football games',
]

vector = CountVectorizer()
print(vector.fit_transform(corpus).toarray())
print(vector.vocabulary_)
```

```
[[0 0 0 1 1 0 1 1 0 1]
 [0 0 0 0 1 1 1 0 1 0]
 [1 1 1 0 1 1 0 1 0 1]]
{'john': 3, 'likes': 4, 'to': 7, 'watch': 9, 'movies': 6, 'mary': 5, 'too': 8,
```

TF-IDF (Term-Frequency-Inverse Document Frequency)

모든 문서에서 자주 등장하는 단어는 중요도가 낮다고 판단하고, 특정 문서에서만 자주 등장하는 단어는 중요도가 높다고 판단하는 것

```
from math import log
import pandas as pd

docs = [
        'John likes to watch movies and Mary likes movies too',
        'James likes to watch TV',
        'Mary also likes to watch football games',
]
```

```
vocab = list(set(w for doc in docs for w in doc.split()))
vocab.sort()

print('단어장의 크기 :', len(vocab))
print(vocab)
```

```
단어장의 크기 : 13
['James', 'John', 'Mary', 'TV', 'also', 'and', 'football', 'games', 'likes', '
```

```
N = len(docs)
N
```

```
3
```

$$idf(d,t) = log\frac{n}{1+df(t)}$$

```
def tf(t, d): # 특정 문서 d에서의 특정 단어 t의 등장 횟수
    return d.count(t)

def idf(t): # 반비례하는 수
    df = 0 #특정 단어 t가 등장한 문서의 수
    for doc in docs:
        df += t in doc
    return log(N/(df+1))+1

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

```
result = []
for i in range(N):
    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_
```

	James	John	Mary	TV	also	and	football	games	likes	movies	to	too	wa
0	0	1	1	0	0	1	0	0	2	2	2	1	
1	1	0	0	1	0	0	0	0	1	0	1	0	
2	0	0	1	0	1	0	1	1	1	0	1	0	

```
result = []
for j in range(len(vocab)):
    t = vocab[j]
    result.append(idf(t))

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

	IDF
James	1.405465
John	1.405465
Mary	1.000000
TV	1.405465
also	1.405465
and	1.405465
football	1.405465
games	1.405465
likes	0.712318
movies	1.405465
to	0.712318
too	1.405465
watch	0.712318

```
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_
```

	James	John	Mary	TV	also	and	football	games	likes
0	0.000000	1.405465	1.0	0.000000	0.000000	1.405465	0.000000	0.000000	1.424636
1	1.405465	0.000000	0.0	1.405465	0.000000	0.000000	0.000000	0.000000	0.712318
2	0.000000	0.000000	1.0	0.000000	1.405465	0.000000	1.405465	1.405465	0.712318

```
'0 : John likes to watch movies and Mary likes movies too',
'1 :James likes to watch TV',
'2 : Mary also likes to watch football games',
```

```
1
```