```
# nltk라는 자연어 처리 패키지를 불러옵니다.
from nltk.tokenize import word_tokenize
import nltk
```

```
# 연습용 데이터 4개를 트레인이라는 리스트안에 작성합니다.
train = [('i like you', 'pos'),
        ('i hate you', 'neg'),
        ('you like me', 'neg'),
        ('i like her', 'pos')]
```

```
# set이라는 집합 타입을 사용하여 중복되는 단어는 제거하고
# 위의 트레인이라는 리스트에서 한 문장씩 가져와
# 단어로 나눠 저장하였습니다.
# 여기서부터 나눠 저장한 단어를 '말뭉치'라고 부르겠습니다.
all_words = set(word.lower() for sentence in train
                for word in word_tokenize(sentence[0]))
all_words
```

```
# 말뭉치들이 저장된 all words와 트레인 안에 있는 4개의 문장과 각각 비교하여
# 말뭉치가 문장에 있으면 True 없으면 False가 저장되도록 하여
# 말뭉치 기준으로 트레인 문장에 속한 단어인지 아닌지를 확인한다.
t = [({word: (word in word_tokenize(x[0])) for word in all_words}, x[1])
    for x in train]
t
```

```
# hate = False              pos : neg    =      1.7 : 1.0
# 이 문장의 뜻은 hate가 False이면 즉, 쓰여있지 않으면 pos(긍정)가 1.7이고 neg(부정)이 1.0이므로 긍
정적인 의미라는 것을 알 수 있습니다.

# like = True               pos : neg    =      1.7 : 1.0
# 그리고 이 문장을 보면 like가 True 즉, 쓰여있으면 pos가 1.7 neg이 1.0으로 긍정적인 의미라는 것을
알 수 있습니다.
classifier = nltk.NaiveBayesClassifier.train(t)
classifier.show_most_informative_features()
```

```
# 이번엔 새로운 테스트 문장을 만든 후
# all words 말뭉치들과 비교하여 있으면 true 없으면 false로 저장시켜줍니다.
test_sentence = 'i like MeRui'
test_sent_features = {word.lower():
                      (word in word_tokenize(test_sentence.lower()))
                      for word in all_words}
test_sent_features
```

```
# 말뭉치들과 비교한 결과를 분류 메소드에 입력하면
# pos라는 긍정이 출력됩니다.
# 왜냐하면 테스트 문장에는 like라는 말뭉치가 있기 때문에 분류기가
# 긍정의 의미로 인식했기 때문입니다.
classifier.classify(test_sent_features)
```

```
# 한글의 형태소 분석을 위한 Twitter을 포함시킨다.
from konlpy.tag import Twitter
pos_tagger = Twitter()
```

```
# 4개의 문장을 긍정과 부정 태그를 넣어 저장합니다.
train = [('메리가 좋아', 'pos'),
        ('고양이도 좋아', 'pos'),
        ('난 수업이 지루해', 'neg'),
         ('메리는 이쁜 고양이야', 'pos'),
         ('난 마치고 메리랑 놀거야', 'pos')]
```

```
# 위의 트레인 문장들을 가지고 단어별로 나눠 말뭉치들을 만듭니다.
# 하지만 트레인 문장들을 말뭉치들로 나눴을 때 고양이도, 고양이야, 메리가, 메리는
# 이라는 단어가 다른 단어로 인식되는 것을 볼 수 있습니다.
all_words = set(word.lower() for sentence in train
                for word in word_tokenize(sentence[0]))
all_words
```

```
# 같은 뜻의 단어지만 다른 단어로 인식하는 문제점을 무시하고
# 트레인의 문장들을 단어로 나눠 말뭉치들과 비교해서 말뭉치가 있으면 true 없으면 false로
# 저장합니다.
t = [({word: (word in word_tokenize(x[0])) for word in all_words}, x[1])
    for x in train]
t
```

```
# 위에서 저장시킨 t를 분석기에 입력해 긍정과 부정의 단어를 학습시킵니다.
classifier = nltk.NaiveBayesClassifier.train(t)
classifier.show_most_informative_features()
```

```
# 긍정의 의미를 가진 테스트 문장을 하나 저장시킵니다.
test_sentence = '난 수업이 마치면 메리랑 놀거야'
```

```
# 그 후 말뭉치들과 테스트 문장을 비교해 값을 저장합니다.
test_sent_features = {word.lower():
                      (word in word_tokenize(test_sentence.lower()))
                      for word in all_words}
test_sent_features
```

```
# 분석기에 위에서 값을 저장한 딕셔너리를 입력하면
# 긍정적인 의미를 가진 문장이 분석기의 결과로는 neg(부정)을 출력시켜 문제가 발생하는 것을
# 볼 수 있습니다.
classifier.classify(test_sent_features)
```

```
# 분석기의 문제를 해결하기 위해
# 한글을 다룰 때는 형태소 분석이 필요하여 단어에 형태소 태그를 붙여주는 tokenize 함수를
# 만들어줍니다.
def tokenize(doc) :
    return ['/'.join(t) for t in pos_tagger.pos(doc, norm=True, stem=True)]
```

```
# 위쪽에서 만들어 놨던 문장 4개를 다시 형태소 태그를 붙인 말뭉치들로 저장시킵니다.
train_docs = [(tokenize(row[0]), row[1]) for row in train]
train_docs
```

```
# 말뭉치를 한 문장씩 나눠 저장해 놓은 train_docs를 모든 단어를 하나씩 토큰에 저장시킵니다.
tokens = [t for d in train_docs for t in d[0]]
tokens
```

```
# 말뭉치에 있는 단어가 비교할 문장에 있는지 없는지를 구분하는 함수를 만듭니다.
def term_exists(doc) :
    return {word: (word in set(doc)) for word in tokens}
```

```
# 문장 4개를 형태소 태그를 붙인 말뭉치들로 저장시킨 train_docs를
# 말뭉치와 비교하는 함수에 입력하여 값을 저장시킵니다.
# 여기서 for d,c 에서 d는 단어이며 c는 긍정, 부정 태그를 뜻합니다.
train_xy = [(term_exists(d), c) for d,c in train_docs]
train_xy
```

```
# 분류기에 긍정, 부정 단어들을 학습 시킵니다.
classifier = nltk.NaiveBayesClassifier.train(train_xy)
```

```
# 다시 테스트 문장을 저장시킵니다.
test_sentence = [("난 수업이 마치면 메리랑 놀거야")]
```

```
# 위에서 한 것과 동일하게 테스트 문장을 단어로 나누고 형태소를 분석합니다.
test_docs = pos_tagger.pos(test_sentence[0])
test_docs
```

```
# 토큰에 저장해 놓은 말뭉치들과 test_docs 단어들과 비교해 존재 여부 값을 저장합니다.
test_sent_features = {word: (word in tokens) for word in test_docs}
test_sent_features
```

```
# 분석기에 위의 저장된 값을 넣으면 pos라는 긍정을 출력시키는 것을 볼 수 있습니다.
classifier.classify(test_sent_features)
```

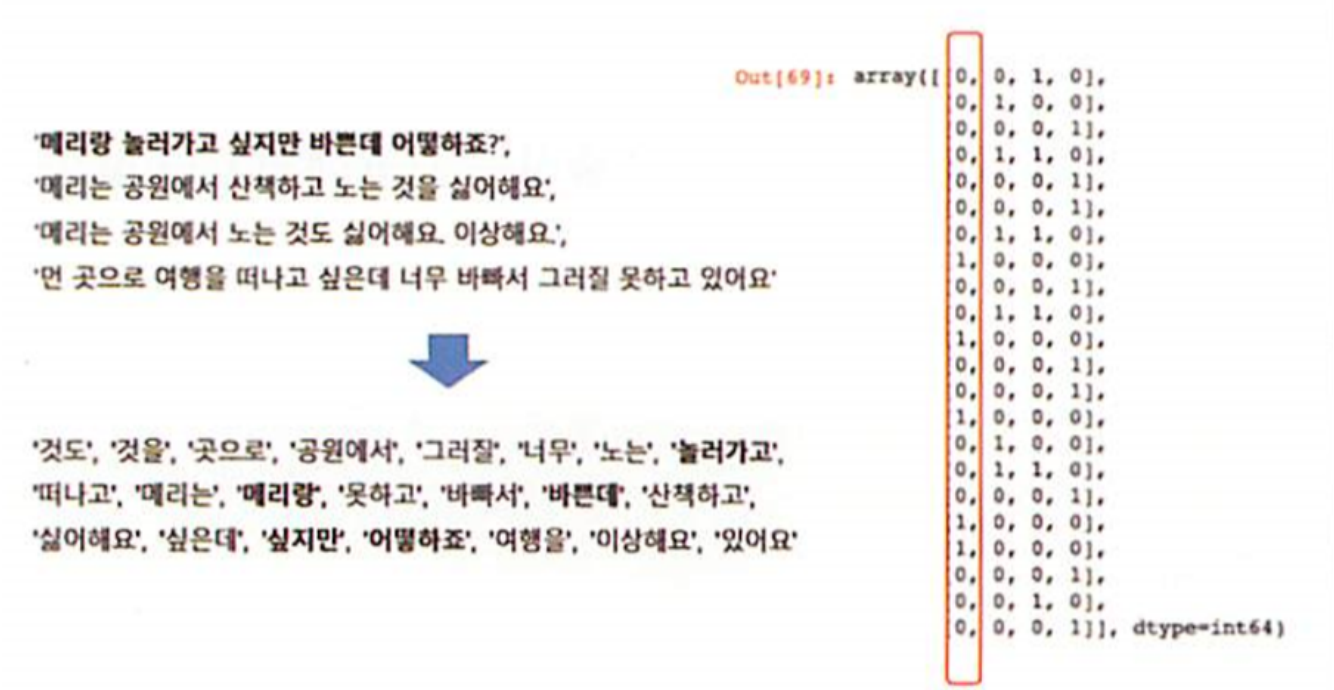
```
# scikit-learn(싸이킷 런)
from sklearn.feature_extraction.text import CountVectorizer
# min_df :
vectorizer = CountVectorizer(min_df = 1)
```

```
# 연습용 문장을 저장합니다.
contents = ['메리랑 놀러가고 싶지만 바쁜데 어떻하죠?',
            '메리는 공원에서 산책하고 노는 것을 싫어해요',
            '메리는 공원에서 노는 것도 싫어해요. 이상해요.',
            '먼 곳으로 여행을 떠나고 싶은데 너무 바빠서 그러질 못하고 있어요']
```

```
# 문장들을 말뭉치들로 나눠 저장했을 때
# 메리랑 메리는이 다른 단어로 구분되는 것을 확인할 수 있습니다.
X = vectorizer.fit_transform(contents)
vectorizer.get_feature_names()
```

```
# 문장은 4개 이며 말뭉치는 22개인 것을 알 수 있습니다.
num_samples, num_features = X.shape
num_samples, num_features
```

```
# 4개의 문장과 22개의 말뭉치들을 비교해 벡터 값으로 나타냅니다.
X.toarray().transpose()
```



여기서 각 열이 각 문장에 대한 벡터 값입니다. 위에서 나눠 저장한 말뭉치들과 비교해 일치하면 1로 일치하지 않으면 0으로 저장합니다.

값을 직접 비교해보자면 첫 번째 열의 8번째 값이 1인데요 이 위치의 말뭉치는 '놀러가고'이고 '놀러가고'라는 말뭉치는 첫 번째 문장에 들어가있습니다. 그리고 첫 번째 행의 11번째 값도 1인데 이 위치의 말뭉치는 메리랑 이기 때문에 1이 출력되게 됩니다. 하지만 여기서 메리랑과 메리는을 다른 단어로 인식 하기 때문에 형태소로 나눠줘야 합니다.

```
# 형태소로 나누기 위해 Twitter을 import 합니다.
from konlpy.tag import Twitter
t = Twitter()
```

```
# 형태소로 문장들을 분석합니다. 분석한 결과 메리랑, 메리는이 메리로 분리해서 같은 단어로 인식하는 것을 볼 수 있습니다.
contents_tokens = [t.morphs(row) for row in contents]
contents_tokens
```

```
# 형태소로 문장들을 분석한 콘텐츠 토큰을 한 문장으로 만들기 위해
# 말뭉치들을 하나씩 이어 붙이는데 말뭉치 사이에는 띄어쓰기를 추가시켜 형태소를 구분합니다.
# 그러면 출력 결과와 같이 문장으로 저장된 것을 확인할 수 있습니다.
contents_for_vectorize = []

for content in contents_tokens :
    sentence = ''
    for word in content :
        sentence = sentence + ' ' + word

    contents_for_vectorize.append(sentence)

contents_for_vectorize
```

```
# 문장들을 말뭉치로 나눠 저장시킵니다.
# 그러면 4개의 문장이 20개의 말뭉치로 나뉜것을 알 수 있습니다.
X = vectorizer.fit_transform(contents_for_vectorize)
num_samples, num_features = X.shape
num_samples, num_features
```

```
# 나뉜 말뭉치들을 확인해봅니다.
vectorizer.get_feature_names()
```

```
# 문장들과 말뭉치들을 비교한 값을 벡터로 나타내봅니다.
X.toarray().transpose()
```

```
# 새로운 문장을 벡터화 시키기 위해 형태소를 분석해 단어별로 저장하고
# 그 단어들을 다시 형태소 기준으로 띄어쓰기를 붙여 한 문장으로 만듭니다.
new_post = ['메리랑 공원에서 산책하고 놀고 싶어요']
new_post_tokens = [t.morphs(row) for row in new_post]

new_post_for_vectorize = []

for content in new_post_tokens :
    sentence = ''
    for word in content :
        sentence = sentence + ' ' + word

    new_post_for_vectorize.append(sentence)

new_post_for_vectorize
```

```
# 문장을 저장되 있는 말뭉치들과 비교하여 값을 벡터화 시킵니다.
new_post_vec = vectorizer.transform(new_post_for_vectorize)
new_post_vec.toarray()
```

```
# 문장의 유사도를 확인하기 위해
# 기존 문장들의 벡터 배열 빼기 새로운 문장의 벡터 배열을 해서
# 델타라는 벡터 배열의 거리를 구합니다.
import scipy as sp
def dist_raw(v1, v2) :
    delta = v1 - v2
    return sp.linalg.norm(delta.toarray())
```

```
# 가장 유사한 문장과 유사도를 저장할 변수를 선언합니다.
best_doc = None
best_dist = 65535
best_i = None
```

```
# 새로운 문장과 4개의 문장들을 하나씩 벡터 길이를 구해주는 함수에 입력합니다.
# 그 후 문장 번호와 벡터 길이 ,문장을 출력시켜 줍니다
# 그리고 가장 유사한 문장의 유사도와 번호를 저장해둡니다.

for i in range(0, num_samples) :
    post_vec = X.getrow(i)
    d = dist_raw(post_vec, new_post_vec)

    print("== Post %i with dist = %.2f : %s" %(i, d, contents[i]))

    if d < best_dist:
        best_dist = d
        best_i = i
```

```
# 가장 유사한 문장의 번호와 유사도 그리고 문장을 출력시킵니다.
print('Best post is %i, dist = %.2f' % (best_i, best_dist))
print('-->', new_post)
print('---->', contents[best_i])
```

```
# 4개의 문장들의 벡터 배열과
# 테스트 문장의 벡터 배열을 출력시켜보면
# 1의 위치가 가장 유사한 것은 위에서 두번째 문장인 것을 확인할 수 있습니다.
for i in range(0, len(contents)) :
    print(X.getrow(i).toarray())

print('-------------------------')
print(new_post_vec.toarray())
```

```
# 각 벡터의 표준을 구해 나눠준 후 거리를 구하도록
# 함수를 만듭니다.
def dist_norm(v1, v2) :
    v1_normalized = v1 / sp.linalg.norm(v1.toarray())
    v2_normalized = v2 / sp.linalg.norm(v2.toarray())

    delta = v1_normalized - v2_normalized

    return sp.linalg.norm(delta.toarray())
```

```
# 새로운 문장과 4개의 문장들을 하나씩 벡터 길이를 구해주는 함수에 입력합니다.
# 그 후 문장 번호와 벡터 길이 ,문장을 출력시켜 줍니다
# 그리고 가장 유사한 문장의 벡터 거리와 번호를 저장해둡니다.
# 벡터 거리가 가장 낮은 문장이 가장 유사한 문장임을 알 수 있습니다.
```

```
best_doc = None
best_dist = 65535
best_i = None

for i in range(0, num_samples):
    post_vec = X.getrow(i)
    d = dist_norm(post_vec, new_post_vec)

    print('== Post %i with dist=%.2f  : %s' %(i, d, contents[i]))

    if d < best_dist :
        best_dist = d
        best_i = i
```

```
# 새로운 문장과 가장 유사한 문장의 번호와 벡터 길이 그리고 문장을 출력시킵니다.
print('Best post is %i, dict = %.2f' % (best_i, best_dist))
print('-->', new_post)
print('--->', contents[best_i])
```

```
# tf는 찾는 문자 t가 d 문자열에서 몇 개인지 세고
# d 라는 문자열의 문자 개수로 나눈 값 입니다.
# idf는 리스트의 문자열 개수를 찾는 문자가 있는 문자열의 개수로 나눈 값 입니다.
def tfidf(t, d, D) :
    tf = float(d.count(t)) / sum(d.count(w) for w in set(d))
    idf = sp.log(float(len(D))/ (len([doc for doc in D if t in doc])))
    return tf, idf
```

```
# 문자열 리스트 3개를 선언하고
# D에 리스트로 저장합니다.
# 그리고 tfidf 함수에 a, b, c 각각 의 문자와 문자열 리스트들을 매개변수로 입력하면
# tf 값과 idf 값이 출력되는 것을 볼 수 있습니다.
a, abb, abc = ['a'], ['a', 'b', 'b'], ['a', 'b', 'c']
D = [a, abb, abc]

print(tfidf('a', a, D))
print(tfidf('b', abb, D))
print(tfidf('a', abc, D))
print(tfidf('b', abc, D))
print(tfidf('c', abc, D))
```

```
# tfidf 계산하는 함수는 싸이킷런에서 TfidfVectorizer를 import해서 사용할 수 있습니다.
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(min_df=1, decode_error='ignore')
```

```
# 4개의 문장을 형태소로 분석하여 단어를 나눠 말뭉치 리스트로 만듭니다.
# 그리고 말뭉치 리스트를 다시 형태소 기준으로 띄어쓰기를 붙여
# 다시 한 문장으로 만들어 저장해 줍니다.
# 그리고 저장되 있는 문장 4개를 벡터화 시킨 뒤,
# 문장의 개수와 나뉜 말뭉치의 개수를 출력 시킵니다.
contents_tokens = [t.morphs(row) for row in contents]

contents_for_vectorize = []

for content in contents_tokens :
    sentence = ''
    for word in content :
        sentence = sentence + ' ' + word

    contents_for_vectorize.append(sentence)

X = vectorizer.fit_transform(contents_for_vectorize)
num_samples, num_features = X.shape
num_samples, num_features
```

```
# 말뭉치들을 출력시켜 봅니다.
vectorizer.get_feature_names()
```

```
# 새로운 문장을 선언하고
# 그 문장을 형태소 별로 분할 시킵니다.
# 그 후 분할 된 단어들을 다시 띄어쓰기를 붙여 한 문장으로 저장시킵니다.
# 그리고 저장된 문장을 출력시켜봅니다.
new_post = ['근처 공원에 메리랑 놀러가고 싶네요.']
new_post_tokens = [t.morphs(row) for row in new_post]

new_post_for_vectorize = []

for content in new_post_tokens :
    sentence = ''
    for word in content:
        sentence = sentence + ' ' + word

    new_post_for_vectorize.append(sentence)

new_post_for_vectorize
```

```
# 저장된 새로운 문장을 벡터화 시킵니다.
new_post_vec = vectorizer.transform(new_post_for_vectorize)
```

```
# 가장 유사한 단어의 번호 및 벡터 거리값을 저장하는 변수를 선언합니다.
# 그리고 4개의 문장의 벡터와 새로운 문장의 벡터의 거리를 계산하여
# 각 문장에 대한 번호, 벡터거리, 문장을 출력 시킵니다.
# 벡터 거리 값이 가장 작은 문장이 가장 유사한 것임을 알 수 있습니다.
best_doc = None
best_dist = 65535
best_i = None

for i in range(0, num_samples) :
    post_vec = X.getrow(i)
    d = dist_norm(post_vec, new_post_vec)

    print('== Post %i with dist = %.2f  : %s' %(i, d, contents[i]))
```

```
# 데이터 프레임을 위한 pandas
# 합계, 평균 등 계산을 위한 numpy
# 운영체제 확인을 위한 platform
# 그래프를 그리기 위한 matplotlib
# 폰트 설정
# url을 다루기 위한 BeautifulSoup, urlopen, urllib
# 시간 함수를 위한 time
import pandas as pd
import numpy as np

import platform
import matplotlib.pyplot as plt

%matplotlib inline

path = "c:/Windows/Fonts/malgun.ttf"
from matplotlib import font_manager, rc
if platform.system() == 'Darwin':
    rc('font', family='AppleGothic')
elif platform.system() == 'Windows':
    font_name = font_manager.FontProperties(fname=path).get_name()
    rc('font', family=font_name)
else:
    print('Unknown system... sorry~~~~')

plt.rcParams['axes.unicode_minus'] = False

from bs4 import BeautifulSoup
from urllib.request import urlopen
```

```
import urllib
import time
```

```
# 접근할 주소를 html에 저장하고
# 첫 번째 글의 데이터를 가져오기 위해 매개변수 num=1로 입력하고 검색할 키워드를 여친 선물로 입력합니다
# 그리고 response의 html 데이터를 soup에 저장시킵니다.
# 그후 태그 'dl'에 있는 html 데이터를 tmp에 저장시킵니다.
tmp1 = 'https://search.naver.com/search.naver?where=kin'
html = tmp1 + '&sm=tab_jum&ie=utf8&query={key_word}&start={num}'

response = urlopen(html.format(num=1, key_word=urllib.parse.quote('여친 선물')))

soup = BeautifulSoup(response, "html.parser")

tmp = soup.find_all('dl')
```

```
# tmp에 저장된 html을 라인별로 태그를 제외한 나머지 내용들만
# tmp_list에 저장시킵니다.
tmp_list = []
for line in tmp:
    tmp_list.append(line.text)

tmp_list
```

```
# for문의 진행 상황을 알 수 있는 tqdm의 tqdm notebook을 import 합니다.
# 위에서는 첫번째 글의 데이터만 가져왔지만 이번에는 천번째까지의 데이터를 100개만 가져올것입니다.
# 그래서 num에 n을 저장시키고, 키워드는 여자친구 선물로 저장시킵니다.
# 그리고 html 데이터를 가져온뒤 dl 태그 안에서 태그 값은 제외하고 내용만 따로 저장시킵니다.
from tqdm import tqdm_notebook

present_candi_text = []

for n in tqdm_notebook(range(1, 1000, 10)):
    response = urlopen(html.format(num=n, key_word=urllib.parse.quote('여자 친구 선물')))

    soup = BeautifulSoup(response, 'html.parser')

    tmp = soup.find_all('dl')

    for line in tmp:
        present_candi_text.append(line.text)

    time.sleep(0.5)
```

```
# 저장된 내용들을 확인합니다.
present_candi_text
```

```
# 문장을 형태소 분석을 하여 말뭉치를 나누기 위해
# nltk 와 Twitter를 import 시켜줍니다.
import nltk
from konlpy.tag import Twitter
t = Twitter()
```

```
# 단어를 한번에 나누기 위해 라인 별로 저장된 내용들을 하나의 글로 저장시킵니다.
present_text = ''

for each_line in present_candi_text[:1000]:
    present_text = present_text + each_line + '\n'
```

```
# 형태소 기분으로 단어를 나눠 저장시키고 확인해봅니다.
tokens_ko = t.morphs(present_text)
tokens_ko
```

```
# 위에서 나눠 저장한 단어들을 nltk.Text 메소드를 사용하여 단어들의 개수를 확인해봅니다.
# 단어 개수를 확인해보면 육만개가 있는 것을 확인할 수 있는데
# 집합 자료형을 사용하여 중복을 제외하면 400개인 것을 확인 할 수 있습니다.
ko = nltk.Text(tokens_ko, name='여자친구 선물')
print(len(ko.tokens))
print(len(set(ko.tokens)))
```

```
# 그리고 vocab 메소드를 사용하여 단어들의 빈도수를 내림차순으로 100개를 출력시켜봅니다.
ko = nltk.Text(tokens_ko, name='여자친구 선물')
ko.vocab().most_common(100)
```

```
# 위의 출력된 리스트를 보면 가, 로, 을, (.)점 등등 불필요한 단어들이 있음을 알 수 있습니다.
# 그래서 불필요한 단어들은 제외하여 다시 저장시킨 뒤, 단어 빈도수를 다시 출력시킵니다.
stop_words = ['.','가','요','답변','...','을','수','에','질문','제','를','이','도',
                    '좋','1','는','로','으로','2','것','은','다',',','니다','대','들',
                    '2017','들','데','..','의','때','겠','고','게','네요','한','일','할',
                    '10','?','하는','06','주','려고','인데','거','좀','는데','~','ㅎㅎ',
                    '하나','이상','20','뭐','까','있는','잘','습니다','다면','했','주려',
```

```
                        '지','있','못','후','중','줄','6','과','어떤','기본','!!',
                        '단어','선물해','라고','중요한','합','가요','....','보이','네','무지']

tokens_ko = [each_word for each_word in tokens_ko
                                                    if each_word not in stop_words]

ko = nltk.Text(tokens_ko, name='여자 친구 선물')
ko.vocab().most_common(50)
```

```
# 단어 빈도수에 대해서 그래프로 그려봅니다.
plt.figure(figsize=(15,6))
ko.plot(50)
plt.show()
```

```
# 워드 클라우드를 그리기 위한 모듈들을 import 합니다.
from wordcloud import WordCloud, STOPWORDS
from PIL import Image
```

```
# 단어의 빈도수 상위 300개를 워드 클라우드로 그려보겠습니다.
# 폰트 설정과, 글씨들의 간격, 배경색 등을 설정하고 워드클라우드에 저장합니다.
# 그리고 그림의 사이즈를 입력하고 imshow 메소드의 매개변수로 워드클라우드를 입력하고
# 그림을 출력시킵니다.
data = ko.vocab().most_common(300)

# for mac : font_path='/Library/Fonts/AppleGothic.ttf'
wordcloud = WordCloud(font_path='c:/Windows/Fonts/malgun.ttf',
                      relative_scaling = 0.2,
                      #stopwords=STOPWORDS,
                      background_color='white',
                      ).generate_from_frequencies(dict(data))
plt.figure(figsize=(16,8))
plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```

```
# 저장되어 있는 하트 그림 파일을 불러와
# 그 위에 워드 클라우드를 그려보겠습니다.
mask = np.array(Image.open('./DataScience/data/09. heart.jpg'))

from wordcloud import ImageColorGenerator

image_colors = ImageColorGenerator(mask)
```

```
# 이번에는 200개의 단어를 하트모양의 클라우드로 그려보겠습니다.
# 폰트 설정과, 글씨들의 간격, 배경색 등을 설정하고 워드클라우드에 저장합니다.

data = ko.vocab().most_common(200)

# for mac : font_path='/Library/Fonts/AppleGothic.ttf'
wordcloud = WordCloud(font_path='c:/Windows/Fonts/malgun.ttf',
                      relative_scaling = 0.1, mask=mask,
                      background_color='white',
                      min_font_size=1,
                      max_font_size=100
                      ).generate_from_frequencies(dict(data))
default_colors = wordcloud.to_array()
```

```
# 그리고 그림의 사이즈를 입력하고 imshow 메소드의 매개변수로 워드클라우드를 입력합니다
# 그림을 출력 시킵니다.
plt.figure(figsize=(12,12))
plt.imshow(wordcloud.recolor(color_func=image_colors), interpolation='bilinear')
plt.axis('off')
plt.show()
```

```
# 조사나 어미를 제거하기 위해 자연어를 처리해주는 word2vec을 import 시킵니다.
import gensim
from gensim.models import word2vec
```

```
# 조사나 어미를 제거하기 위해
# 형태소를 분석하는 트위터를 선언하고
# 조사와 어미를 제거한 결과를 저장할 results를 리스트로 선언합니다.
# 그리고 lines에 이전에 html의 태그를 빼고 내용만 저장해둔 텍스트를 저장합니다.
# for 문에 들어와서 한 라인씩 형태소로 분석하고
# 조사나 어미인 것들은 제외한 나머지 단어들을 리스트 r에 저장합니다.
# 그리고 r에 있는 단어들의 사이를 띄어쓰기로 한 문장으로 붙입니다.
twitter = Twitter()
results = []
lines = present_candi_text

for line in lines:
    malist = twitter.pos(line, norm=True, stem=True)
    r = []

    for word in malist:
        if not word[1] in ["Josa", "Eomi", "Punctuation"]:
            r.append(word[0])
```

```
    r1 = (' '.join(r)).strip()
    results.append(r1)
    print(r1)
```

```
# 위에서 만든 데이터를 저장해놓습니다.
data_file = 'pres_girl.data'
with open(data_file, 'w', encoding='utf-8') as fp :
    fp.write('\n'.join(results))
```

```
# 그리고 데이터를 다시 불러와서 word2vec에 학습시킨뒤 모델을 만들어 줍니다.
data = word2vec.LineSentence(data_file)
model = word2vec.Word2Vec(data, size=200, window=10, hs=1, min_count=2, sg=1)

model.save('pres_girl.model')
model = word2vec.Word2Vec.load('pres_girl.model')
```

```
# 데이터에서 선물과 유사한 단어들 찾아보겠습니다.
model.most_similar(positive=['선물'])
```

```
# 데이터에서 여자친구와 유사한 단어들을 찾아보겠습니다.
model.most_similar(positive=['여자친구'])
```