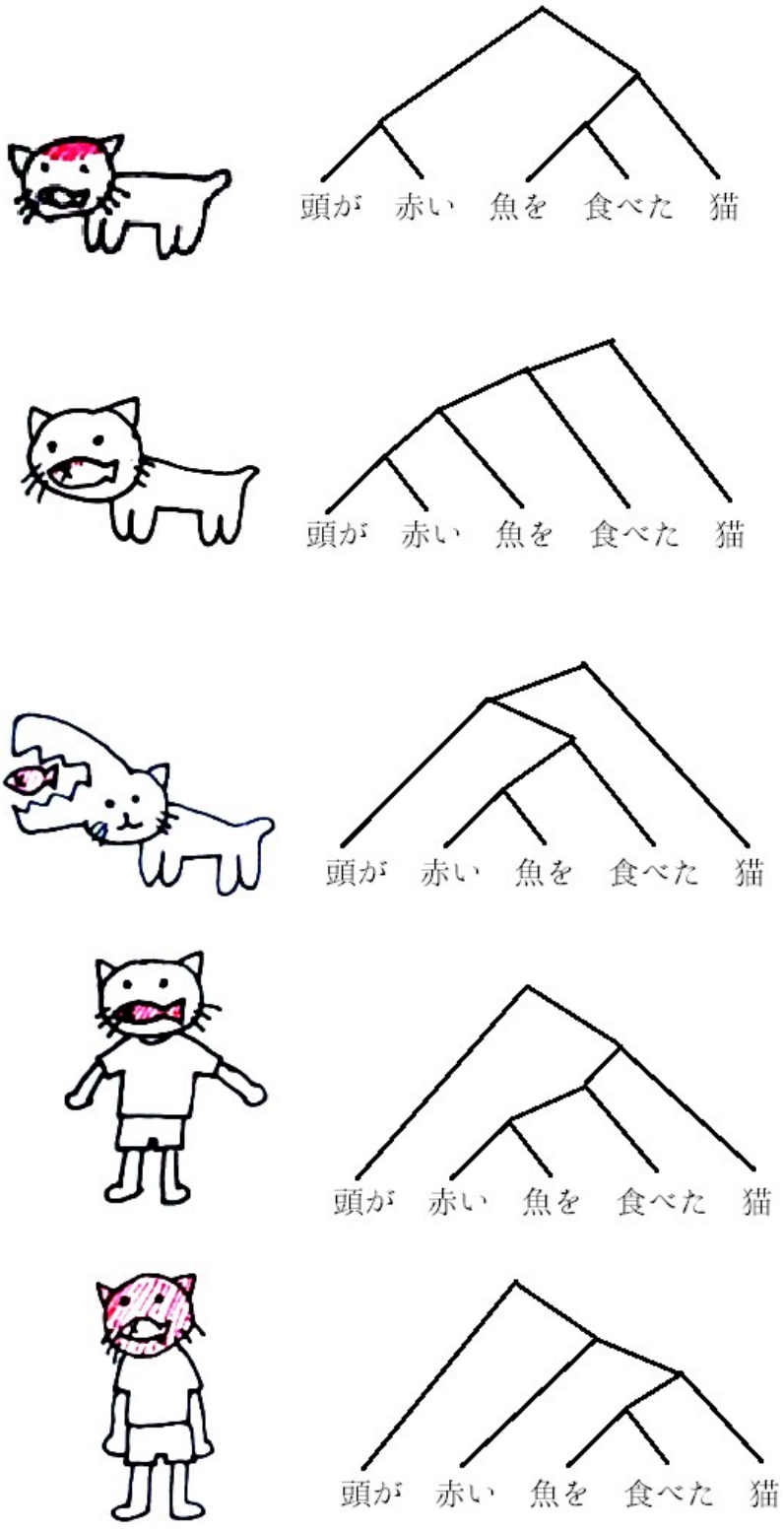
3. 단어 문서 행렬

3.1. 단어 자루 모형

"머리가 빨간 생선을 먹는 고양이(頭が赤い魚を食べた猫)"라는 간단한 표현은 적어도 5가지 뜻으로 해석될 수 있다.



이와 같이 자연어 텍스트는 문장 구조를 어떻게 해석하느냐에 따라 다양한 의미로 해석될 수 있다. 그런데 일반적인 텍스트는 문장 구조가 이렇게까지 복잡하지 않고, 대부분의 경우에는 사용된 단어만 보더라도 대략의 의미를 파악할 수 있다.

따라서 많은 텍스트 분석은 "단어 자루 모형(Bag-of-Words model)"을 사용한다. 간단히 말해 텍스트에서 문장 구조를 무시하고, 오로지 사용된 단어의 종류와 빈도만을 바탕으로 분석하는 것이다.

단어 자루 모형은 앞서 "머리가 빨간 생선을 먹는 고양이"의 경우 단순히 다음 단어들의 조합으로만 본다.

- 명사: 머리, 생선, 고양이
- 용언: 빨갛다, 먹다
- 조사: 가, 을

단어 자루 모형은 다음과 같은 장점이 있다. 먼저, 전처리가 단순하다. 문장을 단순히 단어로 끊기만 하면 되기 때문이다. 영어의 경우 단어 변화가 크지 않기 때문에 띄어쓰기만으로 단어를 구별하기도 한다. 한국어의 경우에는 조사나 어미를 떼내야 하기 때문에 형태소 분석이 필요하지만 그래도 상대적으로 단순한 편이다.

또, 텍스트에 나타난 단어들의 빈도를 세서 간단히 수치화시킬 수 있고, 일반적으로 사용하는 통계 방법론들을 적용하기도 쉽다.

또한 분석 결과를 해석하기도 쉽다. "머리가 빨간 생선을 먹는 고양이"라는 표현은 어쨌든 고양이와 관련된 글이다. 왜냐하면 '고양이'라는 단어가 나왔기 때문이다.

다만 단어 자루 모형은 문장 구조를 무시하기 때문에 "영희가 철수를 사랑했다"와 "철수가 영희를 사랑했다"와 같은 어순 상의 차이를 파악할 수 없다. 또 "어제는 밤에 밤을 구워 먹었다"에서처럼 동음이의어를 맥락으로 구별하기도 어렵다.

3.2. 단어 문서 행렬

단어 문서 행렬(term-document matrix: 이하 TDM)는 문서별로 나타난 단어의 빈도를 표(행렬) 형태로 나타낸 것이다. 아래 두 문장을 보자.

- 문장 1: 분석 중엔 역시 텍스트 분석
- 문장 2: 텍스트 분석 재밌다

위의 두 문장을 TDM으로 나타내면 아래와 같이 나타낼 수 있다.

	분석	재밌다	텍스트
문장 1	2	0	1
문장 2	1	1	1

자연어 텍스트를 TDM으로 바꾸면 일반적인 정형 데이터와 거의 같은 형태가 된다.

	방문 횟수	포인트	구매 건수
고객 1	2	0	1
고객 2	1	1	1

TDM은 자연어 텍스트를 정형 데이터와 같은 방식으로 다룰 수 있게 해준다.

3.2.1. 희소 행렬

그러나 TDM에는 자연어의 특수성에서 비롯된 한 가지 특징이 있다. 자연어에서 사용되는 단어 수만 개가 넘는다. 그러나 하나의 문장이나 문서에서 사용되는 단어의 수는 제한적이다. 따라서 TDM에서 대부분의 칸은 값이 0이 된다. 이런 종류의 행렬을 희소(sparse) 행렬이라고 한다.

컴퓨터에서는 메모리를 아끼기 위해 희소 행렬에 특별한 압축 방식을 사용하기도 한다. 10만개의 문서에서 1만 종의 단어를 표현하기 위해서는 대략 20GB가 필요하다. 그러나 만약 95%가 0이라면 0을 제외하여 150MB 정도로 압축할 수 있다.

TDM은 용량이 크다는 것 외에도 통계적으로도 몇 가지 문제가 있다. 통계에서는 사례 수에 비해 변수의 수가 많으면 분석이 잘 되지 않는다. TDM에서는 단어가 변수의 역할을 하고, 자연어에는 단어의 종류가 많기 때문에 TDM에 단순히 일반적인 통계 기법을 적용하면 분석이 어렵다. 이런 문제에 대처하기 위해 정규화, 단어 임베딩 등 다양한 기법을 적용한다.

3.3. TDM 실습

3.3.1. 데이터 불러오기

데이터를 다운 받는다.

```
from urllib.request import urlretrieve
urlretrieve('http://doc.mindscale.kr/km/unstructured/review.gz', 'review.gz')
```

```
('review.gz', <http.client.HTTPMessage at 0x1be8fa90780>)
```

`pandas` 를 임포트한다. `pandas` 는 표 형태의 데이터를 다루는 파이썬 라이브러리다.

```
import pandas
```

우리가 다룰 데이터는 아마존에서 휴대폰(cell phone)에 대한 고객 리뷰이다. 리뷰 데이터는 CSV

형식으로 되어 있다. CSV 형식은 단순 텍스트 파일로 열과 열 사이를 콤마로 구분한 것이다. CSV는 콤마(comma)로 구분된(separated) 값들(values)의 줄임말이다.

`pandas` 의 `read_csv` 함수는 이름 그대로 CSV 파일을 읽어들인다.

```
df = pandas.read_csv('review.gz')  # 데이터 파일명을 넘겨 읽어들인 CSV 파일을 df 변수에
```

```
df.head()  # df의 첫 5행을 확인한다.
```

	reviewText	overall
0	They look good and stick good! I just don't li...	4
1	These stickers work like the review says they ...	5
2	These are awesome and make my phone look so st...	5
3	Item arrived in great time and was in perfect ...	4
4	awesome! stays on, and looks great. can be use...	5

reviewText 열은 고객들의 리뷰이고 overall 열은 별점이다. 리뷰가 길어 뒷부분은 ...으로 생략되어 있다. 해당 텍스트를 보려면 `.loc` 을 이용해서 본다.

```
df.loc[0, 'reviewText']  # 0행의 reviewText 열
```

```
"They look good and stick good! I just don't like the rounded shape because I was alv
```

`reviewText` 열만 뽑아내려면 아래와 같이 한다.

```
df['reviewText']
```

```
0         They look good and stick good! I just don't li...
1         These stickers work like the review says they ...
2         These are awesome and make my phone look so st...
3         Item arrived in great time and was in perfect ...
4         awesome! stays on, and looks great. can be use...
5         These make using the home button easy. My daug...
...
Name: reviewText, Length: 194439, dtype: object
```

일부 데이터는 리뷰가 없다. 이 경우 빈 칸을 뜻하는 값인 `nan` 으로 채워져있다.

```
df.loc[548, 'reviewText']
```

```
nan
```

그런데 `nan` 은 문자열이 아니기 때문에 후속 처리에 문제가 생길 수 있다. 이 때는 `nan` 을 빈 문자열 `''` 로 바꾸거나, 해당 리뷰의 데이터를 삭제하는 방법이 있다(list-wise deletion). 여기서는 후자의 방법을 사용하겠다. 먼저 df의 형태를 확인해보면 194,439행 2열의 데이터이다.

```
df.shape
```

```
(194439, 2)
```

`dropna` 메소드를 이용해 `nan` 이 포함된 행을 모두 삭제한다.

```
df = df.dropna()
```

다시 형태를 확인해보면 99개 행이 삭제된 것을 알 수 있다.

```
df.shape
```

```
(194340, 2)
```

3.3.2. TDM 만들기

`scikit-learn` 의 `CountVectorizer` 을 불러온다. `scikit-learn` 은 기계학습 라이브러리다.

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

먼저 CountVectorizer를 생성한다. 여러 가지 옵션을 설정할 수 있는데, 여기서는 `max_features` 를 1000으로 설정한다. 이 옵션은 TDM에서 포함시킬 최대(max)의 단어(feature) 수를 말한다. 즉, 빈도 순으로 최대 1000 단어까지만 포함시킨다. `stop_words` 는 분석에서 제외할 불용어를 설정하는 옵션이다. `english` 로 설정하면 영어의 관사, 전치사 등을 자동으로 제외시킨다. 한국어는 지원하지 않기 때문에 한국어를 분석할 때는 다른 방법이 필요하다.

```
cv = CountVectorizer(max_features=1000, stop_words='english')
```

이제 cv를 이용해서 텍스트를 TDM으로 변환한다.

```
tdm = cv.fit_transform(df['reviewText'])
```

`fit_transform` 은 `fit` 과 `transform` 두 개의 메소드를 합친 것이다. `fit` 은 단어 문서 행렬의 형태를 정하는 메소드로, 각 열에 들어갈 단어를 결정한다. 이 정보는 `cv` 의 내부에 저장된다.

`transform` 은 이렇게 정해진 형태로 데이터를 변환하는 메소드이다. 실제 단어 문서 행렬은 `transform` 에 의해 만들어진다. 다른 데이터를 리뷰 데이터와 같은 형태로 변환하고 싶다면, `fit_transform` 이 아닌 `transform` 을 사용한다.

결과적으로 변환된 단어 문서 행렬은 변수 `tdm` 에 할당된다. `tdm` 변수의 형태를 확인해보자.

TDM의 기본 형태

```
tdm.shape
```

```
(194340, 1000)
```

행의 수는 문서의 수이기 때문에 `df` 와 같다. 열의 수는 단어의 수이기 때문에 처음 설정한대로 1000개이다. 앞에서 설명한 대로 단어 문서 행렬의 형태는 `cv` 내부에 저장되기 때문에 단어 목록을 보려면 `cv` 의 `get_feature_names` 메소드를 사용한다.

```
cv.get_feature_names()
```

```
['10',
 '100',
 '12',
 '15',
 '1a',
 ...
 'wouldn',
 'wrong',
 'year',
 'years',
 'yes']
```

다른 방법을 지정해주지 않으면 `CountVectorizer` 는 숫자든 문자든 가리지 않고 공백만으로 단어를 구별한다. 숫자를 제거해주는 것도 가능하지만 자주 나오는 숫자들은 모델명 등 특별히 의미가 있는 숫자인 경우도 있기 때문에 잘 생각해보고 결정해야 한다.

이 단어 목록은 앞으로 자주 쓰게 되므로 `words` 라는 변수에 따로 할당을 해두도록 하자.

```
words = cv.get_feature_names()
```

값 보기

tdm에서 첫 문서의 정보를 보면 Compressed Sparse Row format라는 표현이 나온다. 단어 문서 행렬을 대부분의 값이 0이다. sparse라는 단어는 '희박한'이라는 뜻으로 이렇게 대부분이 0인 행렬을 가리킨다. 이런 희박 행렬을 다룰 때는 메모리를 절약하기 위해 압축된(Compressed) 형식을 사용한다. 그래서 내용을 바로 확인할 수는 없다.

```
tdm[0]
```

```
<1x1000 sparse matrix of type '<class 'numpy.int64'>'
        with 11 stored elements in Compressed Sparse Row format>
```

내용을 확인하려면 압축을 풀어주어야 한다. 이렇게 보면 대부분의 값이 0인 것을 볼 수 있다.

```
doc = tdm[0].toarray()
doc
```

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

이제 문서 `doc` 에서 출현한 단어들을 확인해보자. 여기서 list comprehension이라는 문법을 사용하겠다. 당장은 이 표현을 잘 이해하지 못하더라도 괜찮다. 아래 코드는 주석에 붙인 번호 순서대로 읽으면 된다.

```
[w                          # (4) 단어들의 리스트
 for w, c                   # (2) 각 단어와 빈도 중에
 in zip(words, doc.flat)    # (1) 단어들과 빈도를 짝지운다.
 if c > 0]                  # (3) 빈도가 0보다 큰
```

```
['buy',
 'don',
 'good',
 'just',
 'kept',
 'like',
 'look',
 'product',
 'shape',
 'stick',
 'won']
```

3.3.3. 단어 빈도

이제 전체 코퍼스에서 단어 빈도를 구해보자. 이때는 합계를 구하는 `sum` 메소드를 사용한다. `sum` 메소드에는 `axis` 라는 옵션을 설정할 수 있다. `axis=0` 이면 열별로 모든 행의 합계를 구한다. `axis=1` 이면 행별로 모든 열들의 합계를 구한다. 옵션을 지정하지 않으면 모든 값의 합계를 구한다.

```
count = tdm.sum(axis=0)
```

이제 단어와 단어 빈도를 짝지어 리스트로 만들자. `count` 가 행렬 형태이기 때문에 `.flat` 을 이용해 평평한 형태로 바꿔준다. 그리고 `zip` 을 이용해 단어와 빈도를 하나씩 짝을 짓고, 마지막으로 `list` 를 이용해서 리스트로 변환한다.

```
word_count = list(zip(words, count.flat))
```

이제 내용을 확인해보자.

```
word_count
```

```
[('10', 7810),
 ('100', 4240),
 ('12', 2490),
 ('15', 2309),
 ('1a', 3029),
 ...
 ('wouldn', 4446),
 ('wrong', 4069),
 ('year', 5375),
 ('years', 4820),
 ('yes', 3467)]
```

빈도순 정렬

먼저 operator 모듈을 임포트한다.

```
import operator
```

정렬한다.

```
word_order = sorted(
    word_count,
    key=operator.itemgetter(1), # 0이면 단어 순, 1이면 빈도 순
    reverse=True)               # 내림차순(역순)으로 정렬한다
```

가장 많이 사용된 10개 단어를 확인한다.

```
word_order[:10]
```

```
[('phone', 181717),
 ('case', 150035),
 ('like', 74073),
 ('great', 68898),
 ('use', 63229),
 ('screen', 62373),
 ('just', 61980),
 ('good', 60355),
 ('battery', 59361),
 ('iphone', 49758)]
```

3.3.4. 단어 구름

단어 구름을 그릴려면 `wordcloud` 패키지를 설치해야 한다.

먼저 윈도에서는 Unofficial Windows Binaries for Python Extension Packages에서 wordcloud-1.3.3-cpXX-cpXXm-win_amdXX.whl를 자신의 Python 버전과 운영체제 bit에 맞게 다운로드한 후, 다운로드 폴더에서 명령창을 열어 아래 명령을 입력한다.

```
pip install wordcloud-1.3.3-cpXX-cpXXm-win_amdXX.whl
```

맥과 리눅스에서는 터미널에서 다음과 같이 입력하면 다운로드와 설치가 자동으로 이뤄진다.

```
pip install wordcloud
```

그래프 출력 설정

jupyter notebook에서는 그래프를 그리면 새 창으로 뜬다. 그래프를 노트북과 함께 저장하려면 아래 매직 명령으로 설정을 바꿔준다.

```
%matplotlib inline
```

단어 구름 그리기

wordcloud에서 WordCloud를 임포트 한다. 대소문자에 유의

```
from wordcloud import WordCloud
```

배경색은 흰색, 가로 400픽셀, 세로 300픽셀의 그림을 설정한다.

```
wc = WordCloud(background_color='white', width=400, height=300)
```

word_count를 위의 설정의 그림으로 변환하여 `cloud` 변수에 할당한다.

```
cloud = wc.fit_words(word_count)
```

`cloud` 변수의 그림을 노트북에 붙여 넣으려면 아래와 같이 한다.

```
cloud.to_image()
```

```
<PIL.Image.Image image mode=RGB size=400x300 at 0x1B18A2D7B38>
```


phone
case
like
great
use
good
just
screen
iphone
battery
charger
charging
product
charge
really
works
quality
protector
little
price
nice
does
love
time
easy
device
work
usb
don
fit
used
better
recommend
bought
bluetooth
power
protection
cable
cover
got
hard
buy
did
long
new
phones
need
thing
using
looks
fits
sound
doesn
cases
ve
way
make
right
pretty
samsung
bit
headset
galaxy
want
best
perfect
color
lot
car
feel
look
think
easily
small
plastic
devices
problem
speaker
button
far
34