

# Machine Learning with Python

Working with Text Data

# Contacts

Haesun Park

Email : [haesunrpark@gmail.com](mailto:haesunrpark@gmail.com)

Meetup: <https://www.meetup.com/Hongdae-Machine-Learning-Study/>

Facebook : <https://facebook.com/haesunrpark>

Blog : <https://tensorflow.blog>

# Book

파이썬 라이브러리를 활용한 머신러닝, 박해선.

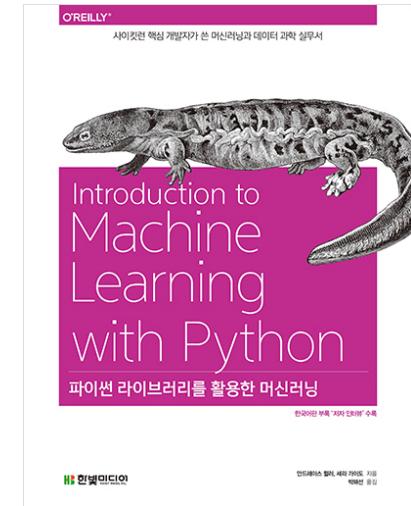
(Introduction to Machine Learning with Python, Andreas Muller & Sarah Guido의 번역서입니다.)

번역서의 1장과 2장은 블로그에서 무료로 읽을 수 있습니다.

원서에 대한 프리뷰를 온라인에서 볼 수 있습니다.

Github:

[https://github.com/rickiepark/introduction\\_to\\_ml\\_with\\_python/](https://github.com/rickiepark/introduction_to_ml_with_python/)



# 문자열 데이터

# 문자열 데이터의 종류

## 범주형 데이터

- \* 고정된 목록: 예) 색깔 드롭다운 박스, “빨강”, “녹색”, “파랑”, “검정”, “검점”

4장 원-핫-인코딩



## 범주에 의미를 연결시킬 수 있는 임의의 문자열

- \* 텍스트 필드: “회색”, “쥐색”, “벨로키랍토르 엉덩이”
- \* 자동 매핑 어려움, 보편적인 값이나 넓은 범주로 정의, “여러가지 색”, “그 외”

## 구조화된 문자열 데이터

- \* 범주에 속하지 않지만 일정한 구조를 가짐: 장소, 이름, 날짜, 전화번호

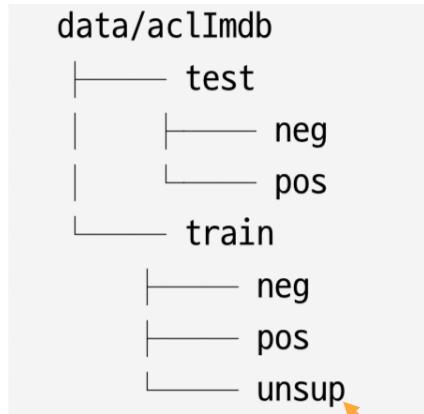
## 텍스트 데이터

- \* 자유로운 문장: 트윗, 리뷰, 소설, 위키

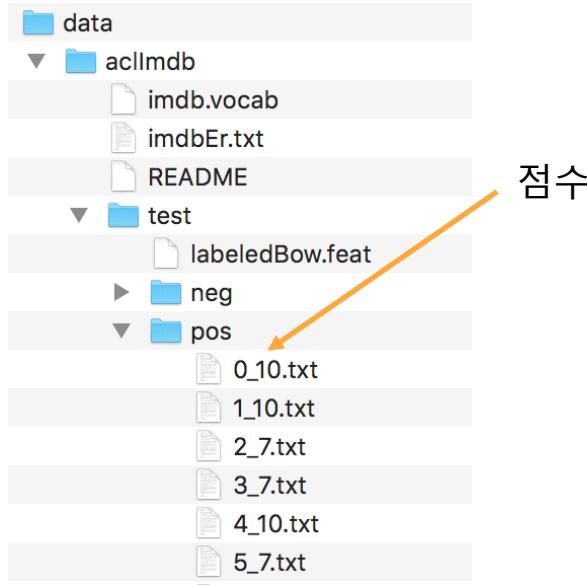
# IMDb 리뷰 데이터셋

스탠퍼드 앤드류 마스 Andrew Mass: <http://ai.stanford.edu/~amaas/data/sentiment/>

IMDb 리뷰: 1~4는 음성(부정평가) vs 7~10은 양성(긍정평가)

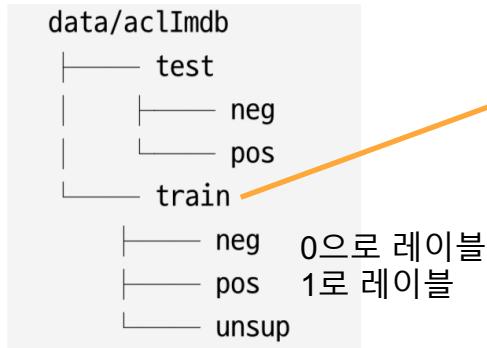


삭제



# 훈련/테스트 데이터

load\_files 함수로 훈련 데이터와 테스트 데이터를 각각 읽어 들임



open(.., "rb")  
Python 3→bytes  
Python 2→str

```
from sklearn.datasets import load_files
reviews_train = load_files("data/aclImdb/train/")
# 텍스트와 레이블을 포함하고 있는 Bunch 오브젝트를 반환합니다.
text_train, y_train = reviews_train.data, reviews_train.target
print("text_train의 타입: {}".format(type(text_train)))
print("text_train의 길이: {}".format(len(text_train)))
print("text_train[6]:\n{}".format(text_train[6]))
```

text\_train의 타입: <class 'list'>  
text\_train의 길이: 25000  
text\_train[6]:  
b"This movie has a special way of telling the story, at first i fou

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

```
print("클래스별 샘플 수 (훈련 데이터): {}".format(np.bincount(y_train)))
```

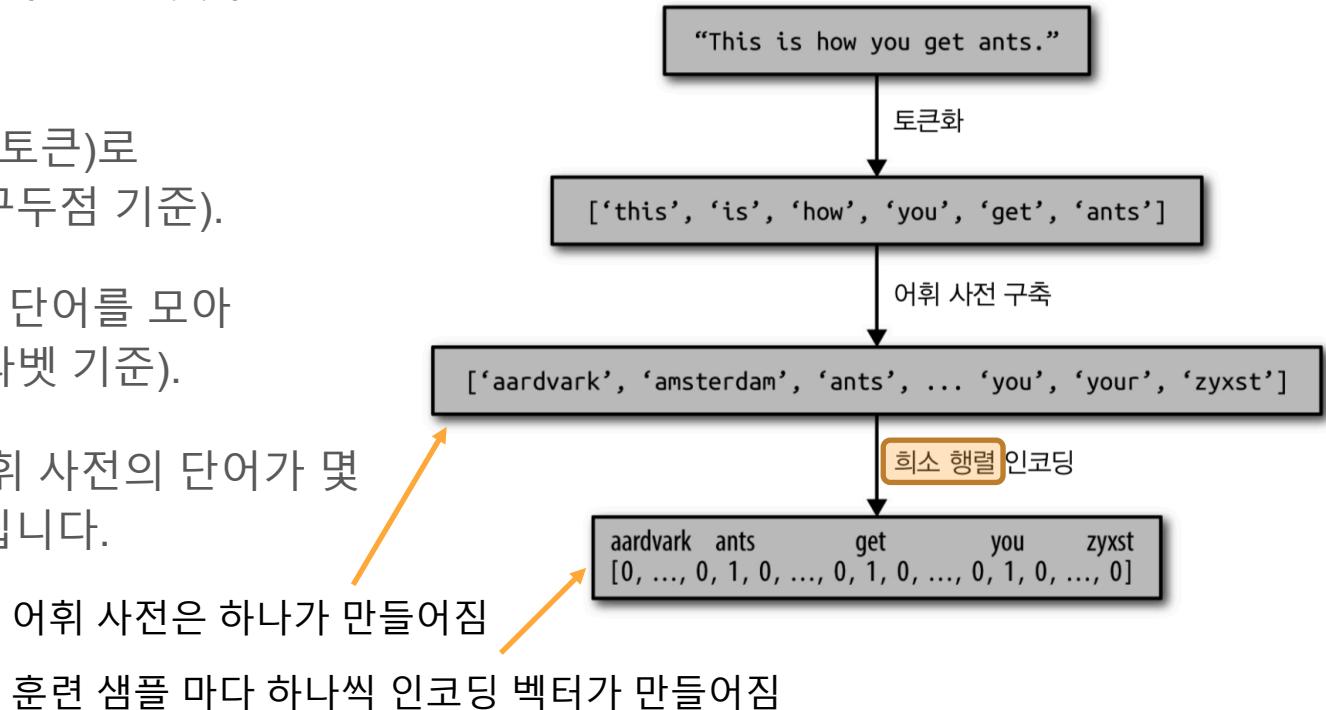
클래스별 샘플 수 (훈련 데이터): [12500 12500]

테스트 데이터의 문서 수: 25000  
클래스별 샘플 수 (테스트 데이터): [12500 12500]

# Bag of Word(BOW)

순서에 상관없이 단어의 출현 횟수를 헤아립니다.

1. 토큰화: 문서를 단어(토큰)로 나눕니다(공백이나 구두점 기준).
2. 어휘 사전 구축: 모든 단어를 모아 번호를 매깁니다(알파벳 기준).
3. 인코딩: 문서마다 어휘 사전의 단어가 몇 번 나타났는지 헤아립니다.



# CountVectorizer

```
bards_words = ["The fool doth think he is wise,",  
               "but the wise man knows himself to be a fool"]
```

```
from sklearn.feature_extraction.text import CountVectorizer  
vect = CountVectorizer()  
vect.fit(bards_words)
```

```
print("어휘 사전의 크기: {}".format(len(vect.vocabulary_)))  
print("어휘 사전의 내용:\n {}".format(vect.vocabulary_))
```

어휘 사전의 크기: 13

어휘 사전의 내용:

```
{'fool': 3, 'be': 0, 'but': 1, 'man': 8, 'himself': 5, 'doth': 2}
```

(2x13)  
희소행렬

```
bag_of_words = vect.transform(bards_words)
```

```
print("BOW의 밀집 표현:\n{}".format(bag_of_words.toarray()))
```

```
BOW의 밀집 표현:  
[[0 0 1 1 1 0 1 0 0 1 1 0 1]  
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

토큰화/어휘사전 구축  
"\b\w\w+\b"

인코딩

실전에서는 메모리 부족  
에러가 날 수 있습니다.

# IMDb's BOW

```
vect = CountVectorizer().fit(text_train)  
X_train = vect.transform(text_train)
```

토큰화/어휘사전 구축  
인코딩

```
feature_names = vect.get_feature_names() ← 단어(특성) 리스트 반환  
print("특성 개수: {}".format(len(feature_names)))  
print("처음 20개 특성:\n{}".format(feature_names[:20]))  
print("20010에서 20030까지 특성:\n{}".format(feature_names[20010:20030]))  
print("매 2000번째 특성:\n{}".format(feature_names[::2000]))
```

특성 개수: 74849

처음 20개 특성:

['00', '000', '000000000001', '00001', '00015', '000s', '001', '003830', '007']

20010에서 20030까지 특성:

['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'draw']

매 2000번째 특성:

['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',

# LogisticRegression + BOW

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
```

```
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("크로스 밸리데이션 평균 점수: {:.2f}".format(np.mean(scores)))
```

크로스 밸리데이션 평균 점수: 0.88

BOW된 훈련 데이터

```
from sklearn.model_selection import GridSearchCV
```

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
```

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
```

```
grid.fit(X_train, y_train)
```

```
print("최상의 크로스 밸리데이션 점수: {:.2f}".format(grid.best_score_))
```

```
print("최적의 매개변수: ", grid.best_params_)
```

CountVectorizer가 포함되지 않았음

최상의 크로스 밸리데이션 점수: 0.89

최적의 매개변수: {'C': 0.1}

```
X_test = vect.transform(text_test)
```

```
print("테스트 점수: {:.2f}".format(grid.score(X_test, y_test)))
```

테스트 점수: 0.88

## min\_df

CountVectorizer는 두 개 이상의 문자나 숫자만(\b\w\w+\b) 토큰으로 분리하여 소문자로 만듭니다.

doesn't → 'doesn', bit.ly → 'bit', 'ly'

soon, Soon, sOon → soon

min\_df: 토큰이 나타날 최소 문서 개수를 지정합니다.

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("min_df로 제한한 X_train: {}".format(repr(X_train)))
```

74849의 1/3

```
min_df로 제한한 X_train: <25000x27271 sparse matrix of type
with 3354014 stored elements in Compressed Sparse
```

# min\_df + LogisticRegression

분류기 성능은 크게 향상되지 않았지만 특성 개수가 줄어들어 속도가 빨라집니다.

```
feature_names = vect.get_feature_names()

print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
```

First 50 features:  
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08'  
Features 20010 to 20030:  
['repentance', 'repercussions', 'repertoire', 'rep  
Every 700th feature:  
['00', 'affections', 'appropriately', 'barbra', 'b  
[ '00', '000', '00000000000001', '00001', '00015', '00  
20010에서 20030까지 특성:  
['dratted', 'draub', 'draught', 'draughts', 'draught  
매 2000번째 특성:  
['00', 'aesir', 'aquarian', 'barking', 'blustering',

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("최적의 크로스 밸리데이션 점수: {:.2f}".format(grid.best_score_))
```

최적의 크로스 밸리데이션 점수: 0.89

# 불용어 stopwords

빈번하여 유용하지 않은 단어로 사전에 정의된 리스트입니다.

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("불용어 개수: {}".format(len(ENGLISH_STOP_WORDS)))
print("매 10번째 불용어:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

불용어 개수: 318  
매 10번째 불용어:  
['none', 'thin', 'are', 'under', 'else', 'next', 'over', 'sometime', 'often

```
# stop_words="english"라고 지정하면 내장된 불용어를 사용합니다.
# 내장된 불용어에 추가할 수도 있고 자신만의 목록을 사용할 수도 있습니다.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("불용어가 제거된 X_train:\n{}".format(repr(X_train)))
```

27271에서  
305개 감소

불용어가 제거된 X\_train:  
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'  
with 2149958 stored elements in Compressed Sparse Row format>

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("최상의 크로스 밸리데이션 점수: {:.2f}".format(grid.best_score_))
```

최상의 크로스 밸리데이션 점수: 0.88

한글 목록은 없습니다

큰 도움이 되지 않음

# max\_df + LogisticRegression

불용어 대신 너무 많이 나타나는 단어를 제외시킵니다.

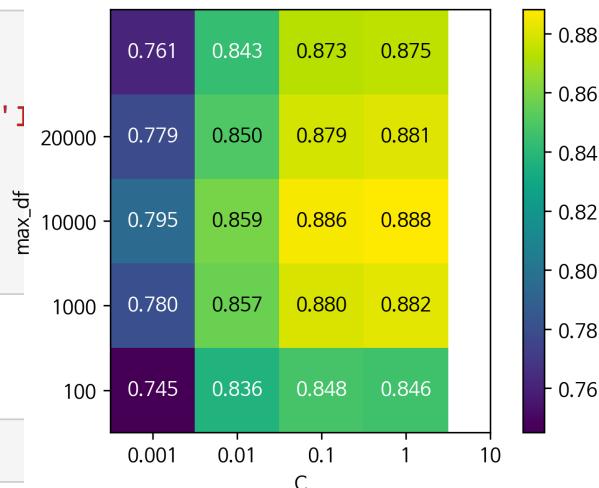
```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(CountVectorizer(), LogisticRegression())
param_grid = {'countvectorizer__max_df': [100, 1000, 10000, 20000], 'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("최상의 크로스 밸리데이션 점수: {:.2f}".format(grid.best_score_))
print(grid.best_params_)
```

최상의 크로스 밸리데이션 점수: 0.89

{'countvectorizer\_\_max\_df': 20000, 'logisticregression\_\_C': 0.1}

```
len(grid.best_estimator_.named_steps['countvectorizer'].vocabulary_)
```

74840



**tf-idf**

# tf-idf

TF-IDF: Term Frequency-Inverse Document Frequency

다른 문서보다 특정 문서에 자주 나타나는 단어는 그 문서를 잘 설명하는 단어라 생각할 수 있으므로 높은 가중치를 줍니다.

다양한 tf-idf 방법 : <https://en.wikipedia.org/wiki/Tf-idf>

$$tfidf(w, d) = tf \times (\log\left(\frac{N + 1}{N_w + 1}\right) + 1)$$

모든 단어를 포함한  
가상의 문서

idf의 최소값은 1

N: 전체 문서 개수,  $N_w$ : 단어 w가 나타난 문서 개수, tf: 단어 빈도수

tf-idf 계산 후에 유클리디안 거리가 1이 되도록 벡터 길이를 바꿉니다.

# TfidfVectorizer

CountVectorizer를 이용하여 BOW를 만들고 TfidfTransformer를 사용합니다.

검증세트의 정보 누설을 막기 위해 파이프라인을 사용하여 그리드서치를 합니다.

CountVectorizer의 매개변수를 지원

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("최상의 크로스 밸리데이션 점수: {:.2f}".format(grid.best_score_))
```

최상의 크로스 밸리데이션 점수: 0.89

# tf-idf가 학습한 단어

(n x word)

word별로  
가장 큰 tfidf

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# 훈련 데이터셋을 변환합니다
X_train = vectorizer.transform(text_train)
# 특성별로 가장 큰 값을 찾습니다
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# 특성 이름을 구합니다
feature_names = np.array(vectorizer.get_feature_names())

print("가장 낮은 tfidf를 가진 특성: \n{}".format(
    feature_names[sorted_by_tfidf[:20]]))

print("가장 높은 tfidf를 가진 특성: \n{}".format(
    feature_names[sorted_by_tfidf[-20:]]))
```

best\_estimator는 파이프라인

가장 낮은 tfidf를 가진 특성:

```
['suplexes' 'gauche' 'hypocrites' 'oncoming' 'songwriting' 'galadriel'
 'emerald' 'mclaughlin' 'sylvain' 'oversee' 'cataclysmic' 'pressuring'
 'uphold' 'thieving' 'inconsiderate' 'ware' 'denim' 'reverting' 'booed'
 'spacious']
```

가장 높은 tfidf를 가진 특성:

```
['gadget' 'sucks' 'zatoichi' 'demons' 'lennon' 'bye' 'dev' 'weller'
 'sasquatch' 'botched' 'xica' 'darkman' 'woo' 'casper' 'doodlebops'
 'smallville' 'wei' 'scanners' 'steve' 'pokemon']
```

전체 문서에 많이: idf→1  
조금씩, 긴문서에만: L2로 tfidf로 작음

# idf가 낮은 단어

전체 문서에 자주 많이 나타난 단어(idf\_ 속성)

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("가장 낮은 idf를 가진 특성:\n{}".format(
    feature_names[sorted_by_idf[:100]]))
```

영화 리뷰의 특징

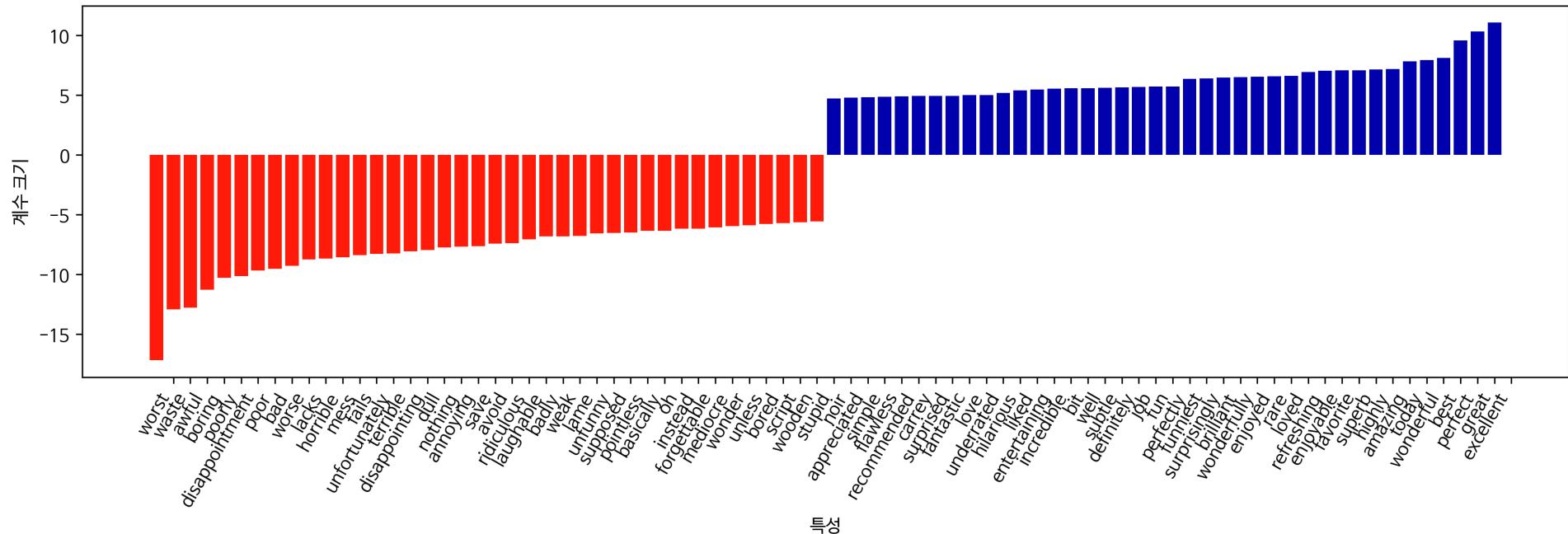
가장 낮은 idf를 가진 특성:

['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'  
'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'  
'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'  
'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'  
'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'  
'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'  
'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'  
'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'  
'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'  
'him']

감성 분석에는 도움이 될지 모르나  
tf-idf 입장에서는 중요도가 떨어짐

## 로지스틱 회귀 모델의 계수

```
grid.best_estimator_.named_steps["logisticregression"].coef_
```



n-그램

# n-Gram

BOW는 단어의 순서를 고려하지 않습니다.

토큰 두 개: 바이그램<sup>bigram</sup>, 토큰 세 개: 트라이그램<sup>trigram</sup>

CountVectorizer, TfidfVectorizer의 ngram\_range 매개변수에 튜플로 최소, 최대 토큰 길이를 지정합니다.

bards\_words:

```
[ 'The fool doth think he is wise,', 'but the wise man knows himself to be a fool' ]
```

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("어휘 사전 크기: {}".format(len(cv.vocabulary_)))
print("어휘 사전:\n{}".format(cv.get_feature_names()))
```

어휘 사전 크기: 14

어휘 사전:

```
[ 'be fool', 'but the', 'doth think', 'fool doth', 'he is', 'h
```

# $n$ -그램의 특징

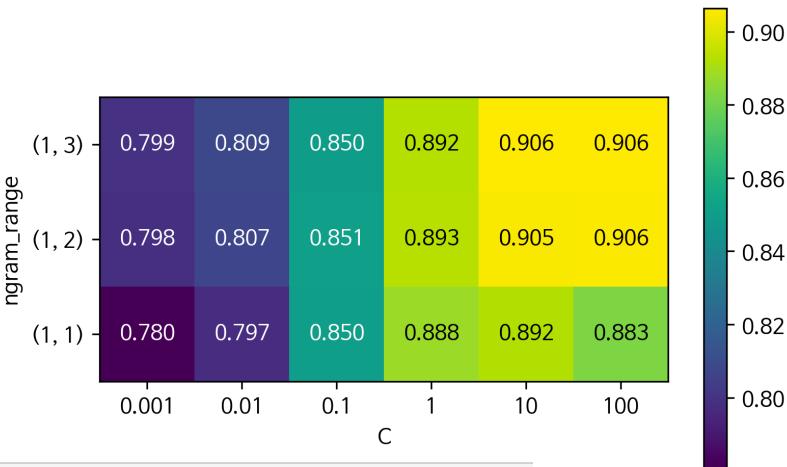
대부분 경우 최소 토큰 길이는 1로 지정합니다.

많은 경우 바이그램을 추가하면 도움이 됩니다.

5-그램까지는 도움이 되지만 특성이 많아져 과대적합될 가능성이 높습니다.

바이그램의 특성 수는 유니그램의 제곱이고, 트라이그램은 유니그램의 세제곱이 됩니다.(중복 순열  $d^n$ )

# n-그램 + TfidfVectorizer



```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# 매개변수 조합이 많고 트라이그램이 포함되어 있기 때문에
# 그리드 서치 실행에 시간이 오래 걸립니다
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100,
                                         "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("최상의 크로스 밸리데이션 점수: {:.2f}".format(grid.best_score_))
print("최적의 매개변수:\n{}".format(
    grid.best_params_))
```

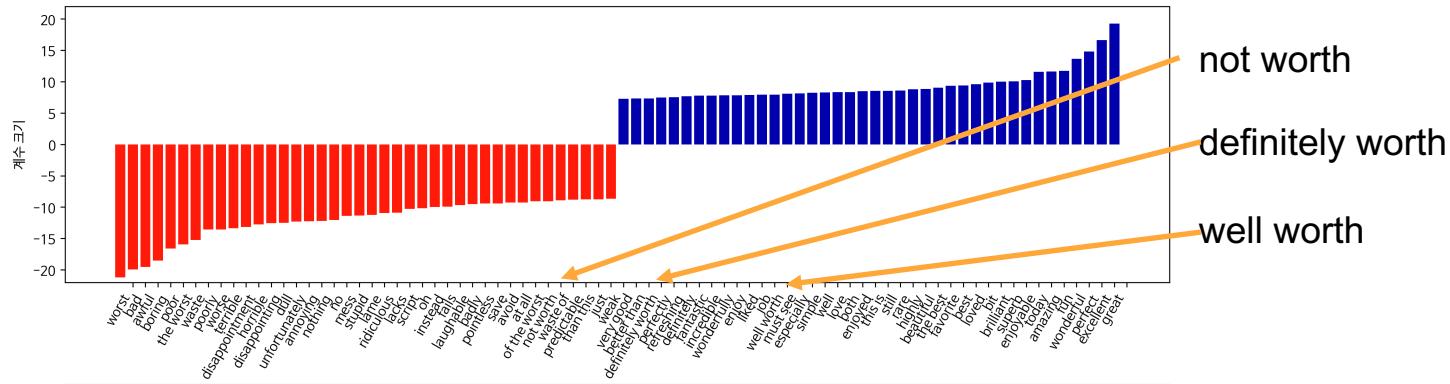
최상의 크로스 밸리데이션 점수: 0.91

최적의 매개변수:

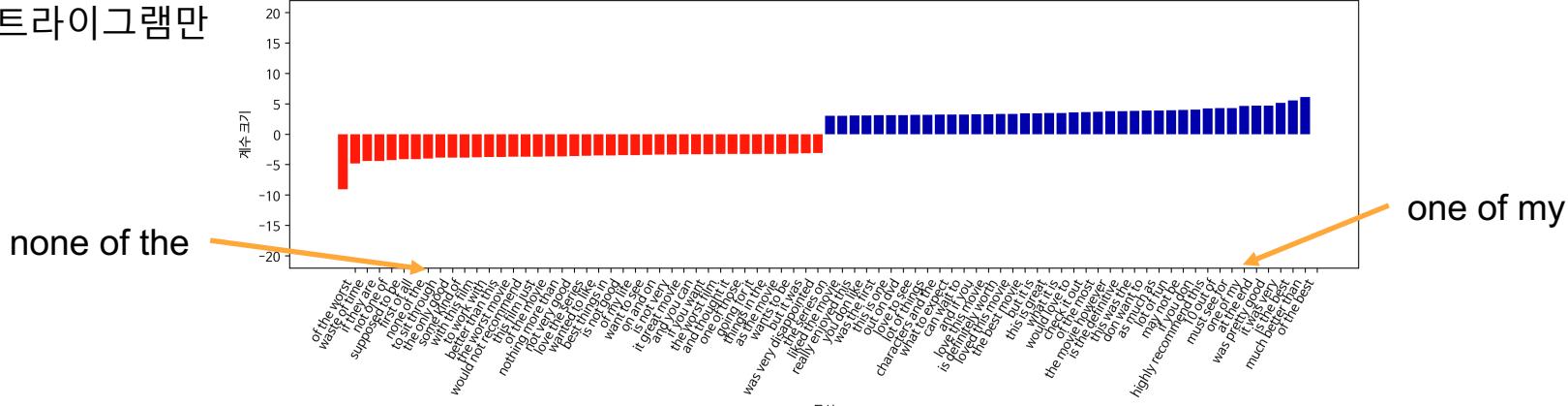
{'tfidfvectorizer\_\_ngram\_range': (1, 3), 'logisticregression\_\_C': 100}

# 로지스틱 회귀의 계수

```
grid.best_estimator_.named_steps['logisticregression'].coef_
```



## 트라이그램만



어간, 표제어

# 어간

BOW의 첫 번째 단계 즉, 토큰화가 중요합니다.

어간 추출<sup>stemming</sup>: 어미(활용될 때 변하는 부분)를 제거 합니다.

“drawback”, “drawbacks”

“보다”, “보니”, “보고”

표제어 추출<sup>lemmatization</sup>(형태소분석): 단어 사전을 활용하고 역할까지 고려합니다.

대표적인 텍스트 처리 패키지: nltk, spacy, gensim

# nltk vs spacy

spacy.tokens.token.Token

```
import spacy
import nltk

# spacy의 영어 모델을 로드합니다
en_nlp = spacy.load('en')
# nltk의 PorterStemmer 객체를 만듭니다
stemmer = nltk.stem.PorterStemmer()

# spacy의 표제어 추출과 nltk의 어간 추출을 비교하는 함수입니다
def compare_normalization(doc):
    # spacy로 문서를 토큰화합니다
    doc_spacy = en_nlp(doc)
    # spacy로 찾은 표제어를 출력합니다
    print("표제어:")
    print([token.lemma_ for token in doc_spacy])
    # PorterStemmer로 찾은 토큰을 출력합니다
    print("어간:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])

    spacy.tokens.doc.Doc
```

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")
```

표제어:

['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 't']

어간:

['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m"]

# CountVectorizer + spacy

```
# 요구사항: CountVectorizer의 정규식 기반 토큰 분할기를 사용하고  
# spacy에서 표제어 추출 기능만 이용합니다.  
# 이렇게 하려고 en_nlp.tokenizer(spacy 토큰 분할기)를  
# 정규식 기반의 토큰 분할기로 바꿉니다  
import re  
# CountVectorizer에서 사용되는 정규식  
regexp = re.compile('(?u)\\b\\w\\w+\\b')  
  
# spacy의 언어 모델을 로드하고 원본 토큰 분할기를 저장합니다  
en_nlp = spacy.load('en')  
old_tokenizer = en_nlp.tokenizer  
# 정규식을 사용한 토큰 분할기를 바꿉니다  
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(  
    regexp.findall(string))  
  
# spacy 문서 처리 파이프라인을 사용해 자작 토큰 분할기를 만듭니다  
# (우리만의 토큰 분할기를 사용합니다)  
def custom_tokenizer(document):  
    doc_spacy = en_nlp(document, entity=False, parse=False)  
    return [token.lemma_ for token in doc_spacy]  
  
# 자작 토큰 분할기를 사용해 CountVectorizer 객체를 만듭니다  
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

```
# 표제어 추출이 가능한 CountVectorizer 객체로 text_train을 변환합니다  
X_train_lemma = lemma_vect.fit_transform(text_train)  
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))  
  
# 비교를 위해 표준 CountVectorizer를 사용합니다  
vect = CountVectorizer(min_df=5).fit(text_train)  
X_train = vect.transform(text_train)  
print("X_train.shape: {}".format(X_train.shape))  
  
X_train_lemma.shape: (25000, 21637)  
X_train.shape: (25000, 27271)
```

# 표제어 추출 + 교차 검증

```
# 훈련 세트의 1%만 사용해서 그리드 서치를 만듭니다
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.99,
                           train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# 기본 CountVectorizer로 그리드 서치를 수행합니다
grid.fit(X_train, y_train)
print("최상의 크로스 밸리데이션 점수 "
      "(기본 CountVectorizer): {:.3f}".format(grid.best_score_))
# 표제어를 사용해서 그리드 서치를 수행합니다
grid.fit(X_train_lemma, y_train)
print("최상의 크로스 밸리데이션 점수 "
      "(표제어): {:.3f}".format(grid.best_score_))
```

최상의 크로스 밸리데이션 점수 (기본 CountVectorizer): 0.721  
최상의 크로스 밸리데이션 점수 (표제어): 0.731

# KoNLPy

# KoNLPy

파이썬 형태소 분석기 래퍼 wrapper

<http://konlpy.org/ko/latest/>

Hannanum, Kkma, Komoran, Mecab, Twitter

네이버 영화 웹 사이트 리뷰 20만개 <https://github.com/e9t/nsmc/>

# 네이버 영화 리뷰 데이터

ratings\_train.txt 와 ratings\_test.txt를 다운로드 받습니다.

nan이 아니고  
빈문자열 그대로

```
df_train = pd.read_csv('data/ratings_train.txt', delimiter='\t', keep_default_na=False)
```

```
df_train.head()
```

	<b>id</b>	<b>document</b>	<b>label</b>
0	9976970	아 더빙.. 진짜 짜증나네요 목소리	0
1	3819312	흠...포스터보고 초딩영화줄....오버연기조차 가볍지 않구나	1
2	10265843	너무재밌었다그래서보는것을추천한다	0
3	9045019	교도소 이야기구먼 ..솔직히 재미는 없다..평점 조정	0
4	6483659	사이몬페그의 익살스런 연기가 돋보였던 영화!스파이더맨에서 늙어보이기만 했던 커스틴 ...	1

# 훈련/테스트 데이터

```
df_test = pd.read_csv('data/ratings_test.txt', delimiter='\t', keep_default_na=False)
text_test = df_test['document'].as_matrix()
y_test = df_test['label'].as_matrix()
```

훈련 데이터와 테스트 데이터의 크기를 확인합니다.

```
len(text_train), np.bincount(y_train)

(150000, array([75173, 74827]))
```

```
len(text_test), np.bincount(y_test)

(50000, array([24827, 25173]))
```

# Twitter 분석기

```
from konlpy.tag import Twitter
twitter_tag = Twitter()

def twitter_tokenizer(text):
    return twitter_tag.morphs(text)
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV

twit_param_grid = {'tfidfvectorizer_min_df': [3, 5, 7],
                   'tfidfvectorizer_ngram_range': [(1, 1), (1, 2), (1, 3)],
                   'logisticregression_C': [0.1, 1, 10]}
twit_pipe = make_pipeline(TfidfVectorizer(tokenizer=twitter_tokenizer), LogisticRegression())
twit_grid = GridSearchCV(twit_pipe, twit_param_grid)

# 그리드 서치를 수행합니다
twit_grid.fit(text_train[0:1000], y_train[0:1000])
print("최상의 크로스 밸리데이션 점수: {:.3f}".format(twit_grid.best_score_))
print("최적의 크로스 밸리데이션 파라미터: ", twit_grid.best_params_)
```

최상의 크로스 밸리데이션 점수: 0.727

최적의 크로스 밸리데이션 파라미터: {'logisticregression\_C': 1, 'tfidfvectorizer\_min\_df': 3, 'tfidf'

```
X_test_twit = twit_grid.best_estimator_.named_steps["tfidfvectorizer"].transform(text_test)
score = twit_grid.best_estimator_.named_steps["logisticregression"].score(X_test_twit, y_test)
print("테스트 세트 점수: {:.3f}".format(score))
```

테스트 세트 점수: 0.721

# Mecab 분석기

```
from konlpy.tag import Mecab
mecab = Mecab()
def mecab_tokenizer(text):
    return mecab.morphs(text)
```

```
mecab_param_grid = {'tfidfvectorizer_min_df': [3, 5, 7],
                     'tfidfvectorizer_ngram_range': [(1, 1), (1, 2), (1, 3)],
                     'logisticregression_C': [0.1, 1, 10, 100]}
mecab_pipe = make_pipeline(TfidfVectorizer(tokenizer=mecab_tokenizer), LogisticRegression())
mecab_grid = GridSearchCV(mecab_pipe, mecab_param_grid, n_jobs=-1)
```

# 그리드 서치를 수행합니다

```
mecab_grid.fit(text_train, y_train)
print("최상의 크로스 밸리데이션 점수: {:.3f}".format(mecab_grid.best_score_))
print("최적의 크로스 밸리데이션 파라미터: ", mecab_grid.best_params_)
```

최상의 크로스 밸리데이션 점수: 0.869

최적의 크로스 밸리데이션 파라미터: {'logisticregression\_C': 10, 'tfidfvectorizer\_min\_df': 3, 'tfid:

```
x_test_mecab = mecab_grid.best_estimator_.named_steps["tfidfvectorizer"].transform(text_test)
score = mecab_grid.best_estimator_.named_steps["logisticregression"].score(x_test_mecab, y_test)
print("테스트 세트 점수: {:.3f}".format(score))
```

테스트 세트 점수: 0.875

# 토익 모델링

# 토픽 모델링 topic modeling

비지도 학습으로 문서를 하나 이상의 토픽에 할당합니다.

뉴스 기사 → “정치”, “스포츠”, “금융”

여기서 토픽은 “주제”를 의미하지 않습니다.

NMF 처럼 어떤 성분에 가깝습니다.

대표적인 토픽 모델링 알고리즘: 잠재 디리클레 할당(LDA)

# LDA

0.19버전에서 n\_components로 변경됨

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

15% 이상 등장하는 단어 제외

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                 max_iter=25, random_state=0)
# 모델 생성과 변환을 한 번에 합니다
# 변환 시간이 좀 걸리므로 시간을 절약하기 위해 동시에 처리합니다
document_topics = lda.fit_transform(X)
```

가장 많이 등장하는 단어 10,000개

```
print("lda.components_.shape: {}".format(lda.components_.shape))
```

```
lda.components_.shape: (10, 10000)
```

# Topics

topic 0	topic 1	topic 2	topic 3	topic 4
between	war	funny	show	didn
young	world	worst	series	saw
family	us	comedy	episode	am
real	our	thing	tv	thought
performance	american	guy	episodes	years
beautiful	documentary	re	shows	book
work	history	stupid	season	watched
each	new	actually	new	now
both	own	nothing	television	dvd
director	point	want	years	got

topic 5	topic 6	topic 7	topic 8	topic 9
horror	kids	cast	performance	house
action	action	role	role	woman
effects	animation	john	john	gets
budget	game	version	actor	killer
nothing	fun	novel	oscar	girl
original	disney	both	cast	wife
director	children	director	plays	horror
minutes	10	played	jack	young
pretty	kid	performance	joe	goes
doesn	old	mr	performances	around

# 요약 및 정리

Natural Language Processing with Python, <http://www.nltk.org/book/>

- spacy: 비교적 최근 출시, 효율적으로 잘 설계됨
- nltk: 기능이 풍부하지만 오래된 라이브러리
- gensim: 토픽 모델링 강점

인공신경망 : word2vec, RNN

감사합니다.