# 빅데이터분석 실습

텍스트 머신러닝 1

데이터 사이언스 전공

담당교수: 곽철완

# 강의 내용

- 문자열 데이터 타입 4가지
- 텍스트 데이터를 BOW로 표현하기
- 불용어 리스트
- 역문헌빈도(tf-idf)
- n-그램 BOW
- 어간 추출과 표제어 추출

# 1. 문자열 데이터 타입

## ▪ 범주형 데이터

- 분석을 위해 수집한 데이터 형태가 몇 종류의 문자열이며
- 각 문자열의 빈도를 통하여 분석하는 데이터 타입
- 예, 좋아하는 색을 묻는 조사에서
  - 응답자는 조사자가 정한 색의 종류(빨강, 녹색, 파랑 등)를 선택하며
  - 데이터 값은 응답자가 선택한 색의 종류 중 하나가 된다
- 하지만, 문자열의 종류가 많다면 범주형 데이터라 하지 않는다

## ▪ 임의의 문자열

- 조사자가 색의 종류를 정해주는 대신에
- 응답자가 색의 종류 문자열을 직접 입력하는 경우
  - 대부분 일반적인 색의 종류 문자열을 기입하지만,
  - 일부는 철자가 틀리거나, 약간 다른 표현을 할 수 있다(예, 회색을 쥐색으로)
- 매우 다양한 문자열이 나열되어 색을 자동으로 매핑하는 것이 불가능하다
- 이 경우, 해당 색을 표현하는 가장 대표적인 단어열(통제어휘)을 선택하고, 유사 단어들은 선택한 대표적인 단어열로 변환시켜 처리한다
  - 통제어휘 리스트: 시소러스라 하며 특정 주제를 나타내는 대표적인 단어(통제어휘)와 유사 단어들을 모아 놓은 리스트이다

## ■ 구조화된 문자열

- 입력 데이터가 일정한 구조를 가지고 있는 것
- 예를 들어, 학번, 이름, 학과, 주소, 전화번호 등이 있다

## ▪ 텍스트 데이터

- 자유로운 형태의 절과 문장으로 구성된 데이터
- SNS, 웹사이트, 전자책 등을 포함한다
- 데이터 세트: 말뭉치 corpus
- 데이터 포인트: 문서 document

# 2.0 예제 어플리게이션

## ▪ 기본 라이브러리

◦ 분석을 위해 기본 라이브러리 import

```
%matplotlib inline
from IPython.display import display
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
import sklearn
```

## ■ 매직 컨맨드

- 설치된 모듈, import한 모듈의 버전 정보, 현재 OS 버전 출력 등
- 아나콘다 프롬프트에서

```
pip install watermark
```

- 쥬피터 노트북에서

```
%load_ext watermark
%watermark –v –p sklearn, numpy, scipy, matplotlib
```

```
%load_ext watermark
%watermark -v -p sklearn,numpy,scipy,matplotlib

CPython 3.7.3
IPython 7.4.0

sklearn 0.20.3
numpy 1.16.2
scipy 1.2.1
matplotlib 3.0.3
```

# 2.1 데이터 세트 삽입

## ▪ 데이터 세트 다운로드

- 사용할 데이터 세트: IMDb 웹사이트에서 수집한 영화 리뷰
  - https://ai.stanford.edu/~amaas/data/sentiment/ 에서 'Large Movie Review Dataset v.10'을 개인 컴퓨터에 다운 받는다
  - 데이터 세트는 aclImdb_v1.tar.gz(철자 주의) 으로 다운 받은 후에 압축을 풀어야 하는데, 압축을 푸는데 최소 1시간 이상 소요된다(압축해제에 7-zip 사용)
  - 데이터 세트가 커서 쥬피터 노트북에서 분석하는 데 시간이 많이 소요되어서 데이터 일부만 사용하는 것을 추천
    - 파일은 텍스트 형식으로 각 폴더에 저장되어 있는데, 각 폴더에서 300개 정도 파일만 각각 복사해서 다른 폴더로 만들어 사용한다

## ▪ 데이터 세트 불러오기

```
from sklearn.datasets import load_files
reviews_train = load_files("E:/work/aclImdb/train")
text_train, y_train = reviews_train.data, reviews_train.target
print("text_train의 타입:", type(text_train))
print("text_train의 길이:", len(text_train))
print("text_train[1]:\n", text_train[1])
```

- load_files 함수를 이용하여 데이터 세트 불러오기(파일이 텍스트 문서 형식)
  - load_files(저장한 데이터 세트 디렉토리)

```python
from sklearn.datasets import load_files
reviews_train = load_files("E:/work/aclImdb/train")
text_train, y_train = reviews_train.data, reviews_train.target
print("text_train의 타입:", type(text_train))
print("text_train의 길이:", len(text_train))
print("text_train[1]:\n", text_train[1])
```

```
text_train의 타입: <class 'list'>
text_train의 길이: 800
text_train[1]:
 b'I had seen Marion Davies in a couple of movies and really couldn₩'t understand her appeal. She cou
attempt to sing and as for her acting - she seemed in a trance. But I hadn₩'t seen her silent comedies
than kidding her own image, as has been suggested here, to me it seems a satire on Gloria Swanson, who
s, went on to highly emotional women₩'s pictures and did end up marrying a Count. Marion, a top mimic,
henever she wanted to be seen as grand, that was Gloria Swanson spot on!!!<br /><br />Colonel Pepper
way from Georgia to Hollywood, determined to prove that his daughter, Peggy, (Marion Davies) will be t
```

HTML 줄바꿈 태그 (<br />) 를 포함하고 있어 제거 필요

◦ 줄바꿈 태그 제거

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

◦ 클래스별 샘플 수 확인

```
print(“클래스 별 샘플 수(학습용 데이터):”, np.bincount(y_train)
```

```
print("클래스별 샘플 수 (학습용 데이터):", np.bincount(y_train))
클래스별 샘플 수 (학습용 데이터): [400 400]
```

◦ 원 데이터 세트의 학습용 데이터는 [12500 12500]이다

# 2.2 텍스트 데이터 BOW로 표현하기

## ■ 데이터 세트 불러오기

- BOW(bag of words)
  - 단어의 출현 빈도를 계산하여 분석
- 빈도 계산 3 단계


"This is how you get ants"

['This', 'is', 'how', 'you', 'get', 'ants']

## ▪ 샘플 데이터에 BOW 적용

◦ bards_words 객체를 만든다

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

◦ CountVectorizer 함수를 이용하여 BOW 구현

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

◦ .fit 메서드는 학습용 데이터를 토큰으로 나누고 어휘사전을 구축하여 vocabulary_ 속성에 저장

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

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

print("어휘 사전의 크기:", len(vect.vocabulary_))
print("어휘 사전의 내용:\n", vect.vocabulary_))

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

```
어휘 사전의 크기: 13
어휘 사전의 내용:
 {'the': 9, 'fool': 3, 'doth': 2, 'think': 10, 'he': 4, 'is': 6, 'wise': 12, 'but': 1, 'man': 8, 'knows': 7,
'himself': 5, 'to': 11, 'be': 0}
```

◦ 학습용 데이터에 대해 BOW 표현을 만들려면 transform 메서드를 호출한다

```
bag_of_words = vect.transform(bards_words)
print("BOW:", repr(bag_of_words)
```

```
bag_of_words = vect.transform(bards_words)
print("BOW:", repr(bag_of_words))
```

```
BOW: <2x13 sparse matrix of type '<class 'numpy.int64'>'
        with 16 stored elements in Compressed Sparse Row format>
```

◦ BOW 표현은 0이 아닌 값만 저장하는 희소 행렬로 저장되어 있다

◦ 이 행렬의 크기는 2 x 13 이며 각 행은 하나의 데이터 포인트를 나타낸다

◦ 각 피처(변수)는 어휘 사전에 있는 각 단어에 대응한다

- 희소 행렬의 실제 내용을 보기 위해 toarray 메서드를 사용한다

```
print("BOW의 밀집 표현:\n", bag_of_words.toarray())
```

```
print("BOW의 밀집 표현:\n", 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]]
```

- 각 단어의 출현 빈도는 0 아니면 1이다(bards_words 각각 문자열은 모두 다른 단어로 구성되어 0개이거나 1개이기 때문이다)
- 첫번째 문자열 ("The fool doth think he is wise,")에서 어휘 사전의 첫 번째 단어 'be'가 0번, 두번째 단어 'but' 도 0번, 세번째 단어 'doth'는 1번 출현한다(15쪽 참조)

## ▪ 영화 리뷰에 대한 BOW

◦ IMDb 의 학습용 데이터세트와 평가용 데이터세트를 문자열 리스트로 바꾸었다 (text_train, text_test)

```
vect = CountVectorizer( ).fit(text_train)
X_train = vect.transform(text_train)
print(“X_train:\n”, repr(X_train))
```

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n", repr(X_train))

X_train:
 <800x13762 sparse matrix of type '<class 'numpy.int64'>'
        with 102834 stored elements in Compressed Sparse Row format>
```

◦ 학습용 데이터의 BOW 표현인 X_train의 크기 800x13762으로, 어휘사전은 단어 13,762개 포함하고 있다(데이터는 SciPy 희소행렬로 저장)

◦ get_feature_names 메서드는 각 피처(변수)에 해당하는 단어를 리스트로 반환한다

```
feature_names = vect.get_feature_names( )
print(“피처 개수:”, len(feature_names))
print(“처음 20개 피처:\n”, feature_names[:20])
print(“10010에서 10030까지 피처:\n”, feature_names[10010:10030])
print(“매 1000번째 피처:\n”, feature_names[::1000])
```

```
feature_names = vect.get_feature_names()
print("피처 개수:", len(feature_names))
print("처음 20개 피처:\n", feature_names[:20])
print("10010에서 10030까지 피처:\n", feature_names[10010:10030])
print("매 1000번째 피처:\n", feature_names[::1000])
```

```
피처 개수: 13762
처음 20개 피처:
 ['00', '000', '0079', '0080', '0083', '00pm', '00s', '06', '08', '10', '100', '1000', '102', '105', '10th',
'11', '1100', '112', '12', '120']
10010에서 10030까지 피처:
 ['relentless', 'relevance', 'relevant', 'reliable', 'relied', 'relief', 'relies', 'relieved', 'religion', 'r
eligious', 'reliquary', 'relish', 'reload', 'reluctant', 'reluctantly', 'rely', 'relying', 'remain', 'remaind
er', 'remaindered']
매 1000번째 피처:
 ['00', 'awfully', 'catches', 'crotch', 'el', 'frequently', 'hum', 'latter', 'moth', 'phillip', 'relative',
'showtime', 'survive', 'updated']
```

- 어휘사전의 앞에서 20번까지 항목을 보면, 17개가 숫자이다
- 영화 리뷰에서 숫자는 특별한 의미를 가지고 있지 않다고 생각한다(007은 제외)
- 또한 relevance와 relevant, religion과 religious는 명사와 형용사이다
- 이들 단어들을 다른 피처로 간주하여 개별적으로 기록하는 것은 적합하지 않다

- 분류기의 성능 수치 계산
  - 피처 추출 방법을 개선하기 전에 현재 모델의 성능을 살펴본다
  - y_train의 학습용 레이블과 X_train의 학습용 데이터의 BOW로 분류기 학습
  - LogisticRegression 모델을 사용하여 성능 측정

```
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)))
```

```
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)))
```

```
C:₩Users₩₩in 2₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩
r will be changed to 'lbfgs' in 0.22. Specify a solver to silenc
  FutureWarning)
```

```
크로스 밸리데이션 평균 점수: 0.82
```

- 교차 검증 평균 점수로 82%는 괜찮은 이진 분류 성능이다
- LogisticRegression의 규제 매개변수 C를 바꿔가면서(그리드 서치 사용) 조정해 본다

```
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(“크로스 밸리데이션 평균 점수:{:.2f}”.format(grid.best_score_))
print(“최적의 매개변수:”, grid.best_params_)
```

- C=0.1에서 교차 검증 점수: 82%
- 이 매개변수를 이용하여 평가용 데이터 세트의 일반화 성능을 측정함(다음 페이지)

```
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_)
```

```
C:₩Users₩win 2₩Anaconda3₩lib₩site-packages₩sklearn₩linear_mod
r will be changed to 'lbfgs' in 0.22. Specify a solver to sil
  FutureWarning)

최상의 크로스 밸리데이션 점수: 0.82
최적의 매개변수:  {'C': 0.1}
```

```
X_test = vect.transform(text_test)
print(“테스트 점수: {:.2f}”.format(grid.score(X_test, y_test)))
```

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

```
테스트 점수: 0.80
```

◦ 일반화 성능 점수는 80%로 괜찮다

- 단어 추출 방법 개선

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

  - 정규표현식은 "\b\w\w+\b"이다
    - \b 경계 구분 \w 둘 이상의 문자나 숫자가 연속된 단어를 찾는다
    - 한 글자로 된 단어는 찾지 않는다
    - doesn't 축약형은 분리된다
    - h8ter는 한 단어로 매칭된다
    - 모든 단어는 소문자로 바꾼다
  - min_df= 매개변수로 토큰이 나타날 최소 문서 개수를 지정(여기서는 5개)

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

```
min_df로 제한한 X_train: <800x2986 sparse matrix of type '<class 'numpy.int64'>'
        with 85550 stored elements in Compressed Sparse Row format>
```

- 피처 수가 13,762개에서 2,986개로 줄었다
- 토큰 내용을 확인한다.

```
feature_names = vect.get_feature_names( )

print("First 50 features:\n", feature_names[:50])
print("Features 2010 to 2030:\n", feature_names[2010:2030])
print("Every 500th feature: \n", feature_names[::500])
```

```
feature_names = vect.get_feature_names()

print("First 50 features:\n", feature_names[:50])
print("Features 2010 to 2030:\n", feature_names[2010:2030])
print("Every 500th feature:\n", feature_names[::500])
```

```
First 50 features:
 ['000', '10', '100', '11', '12', '13', '13th', '14', '15', '16', '17', '1928', '194
5', '1972', '1983', '20', '2001', '2003', '25', '30', '35', '3d', '3rd', '40', '50',
'60', '60s', '70', '70s', '80', '80s', '90', '99', 'abiding', 'ability', 'able', 'ab
out', 'above', 'absence', 'absolute', 'absolutely', 'absurd', 'abuse', 'abusive', 'a
cademy', 'accent', 'accents', 'accept', 'accepted', 'accident']
Features 2010 to 2030:
 ['presented', 'pretentious', 'pretty', 'preview', 'previous', 'previously', 'pric
e', 'prime', 'principal', 'print', 'prior', 'prison', 'private', 'probably', 'proble
m', 'problems', 'produce', 'produced', 'producer', 'producers']
Every 500th feature:
 ['000', 'close', 'filled', 'lee', 'powerful', 'structure']
```

- 희귀한 단어, 혹은 동일 단어의 명사형 및 형용사형이 많이 제거되었다(아직 남아 있다)

- 그리드 서치를 이용한 모델 성능 확인

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

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

```
C:₩Users₩win 2₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model
ure₩arning: Default solver will be changed to 'lbfgs' in 0.22.
ilence this warning.
  Future₩arning)
```

```
최적의 크로스 밸리데이션 점수: 0.81
```

  - 점수는 80%에서 81%로 약간 상승(처리 속도도 빨라지고 불필요한 피처 제거)

# 2.3 불용어

## ▪ 불용어 리스트 사용

◦ 사이킷런의 feature_extraction.text 모듈에 영어 불용어 리스트가 있다

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

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

```
불용어 개수: 318
매 10번째 불용어:
 ['this', 'couldnt', 'even', 'anywhere', 'inc', 'enough', 'every', 'such', 'become',
'how', 'are', 'as', 'three', 'at', 'from', 'i', 'might', 'what', 'etc', 'sixty', 'd
o', 'am', 'yourself', 'here', 'together', 'whose', 'under', 'co', 'fifteen', 'wherea
fter', 'without', 'last']
```

- 불용어를 제외하고 데이터 세트 피처 확인

```
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("불용어가 제거된 X_train:\n", repr(X_train)
```

  - stop_words="english"은 내장된 불용어 사용 지시

```
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("불용어가 제거된 X_train:\n", repr(X_train))
```

```
불용어가 제거된 X_train:
 <800x2723 sparse matrix of type '<class 'numpy.int64'>'
        with 48774 stored elements in Compressed Sparse Row format>
```

  - 피처 수가 2,986에서 2,723으로 감소하였다

- 그리드 서치를 이용하여 교차 검증 점수 확인

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

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

```
C:₩Users₩₩in 2₩Anaconda3₩lib₩site-packages₩sklearn₩linear_modelt
ure₩arning: Default solver will be changed to 'lbfgs' in 0.22. S
ilence this warning.
  Future₩arning)
```

```
최적의 크로스 밸리데이션 점수: 0.86
```

  - 점수가 81%에서 86%로 상승하였다

# 2.4 tf-idf 로 데이터 스케일 변경하기

## ■ 역문헌빈도 tf-idf

- term frequency-inverse document frequency(tf-idf)
  - 말뭉치의 다른 문서보다 특정 문서에 자주 나타나는 단어에 높은 가중치를 주는 방법
  - 특정 문서에 자주 나타나고 다른 문서에는 자주 나타나지 않는 단어는 그 특정 문서를 잘 설명하는 단어라 할 수 있다
  - TfidfVectorizer는 CountVectorizer가 만든 희소행렬을 입력 받아 변환한다
    - TfidfVectorizer는 텍스트 데이터를 입력 받아 BOW 피처 추출과 tf-idf 변환을 수행한다
    - N: 학습용 세트의 문서 개수
    - $N_w$: 단어 w가 나타난 학습용 세트의 문서 개수
    - tf(단어빈도수): 단어 w가 대상문서 d에 나타난 횟수

$$tfidf(w, d) = tf(\log(\frac{N+1}{Nw+1}) + 1)$$

```
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(“최적의 크로스 밸리데이션 점수:{:.2f}”.format(grid.best_score_))
```

- tf-idf 를 적용하여 성능의 86%에서 85%로 감소
- tf-idf 변환은 문서를 구별하여 단어를 찾는 방법이지만, 비지도 학습이다
- 긍정적 리뷰와 부정적 리뷰 레이블이 꼭 관계가 있지 않다는 것을 생각하는 것이 중요하다

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

```
C:₩Users₩₩in 2₩Anaconda3₩lib₩site-packages₩sklearn₩linear_mod
ure₩arning: Default solver will be changed to 'lbfgs' in 0.22
ilence this warning.
  Future₩arning)
```

```
최적의 크로스 밸리데이션 점수: 0.85
```

```
vectorizer = grid.best_estimator_.named_steps[“tfidfvectorizer”]
X_train =vectorizer.transform(text_train)
max_value = X_train.mac(axis=0).toarray( ).ravel( )
sorted_by_tfidf = max_value.argsort( )
feature_names = np.array(vectorizer.get_feature_names( ))

print(“가장 낮은 tfidf를 가진 피처:\n”, feature_names[sorted_by_tfidf[:20]])
print(“가장 높은 tfidf를 가진 피처:\n”, feature_names[sorted_by_tfidf[-20:]])
```

- 학습용 데이터 세트를 vectorizer.transform( ) 함수를 이용하여 변환
- 피처 별로 가장 큰 값을 찾은 후, 순서대로 배열
- 피처 이름을 파악한다

- tf-idf가 낮은 피처는 전체 문서에 걸쳐 많아 나타나거나, 조금씩만 사용되거나, 매우 긴 문서에서만 사용된다
- 여기서는 tf-idf가 높은 피처는 특정 쇼나 드라마에 대한 리뷰에서 많이 나타나지만, 특정 리뷰에 많이 나타나는 경향이 있다(예, zombie, 영화 제목 europa )

```
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 피처:\n",
      feature_names[sorted_by_tfidf[:20]])

print("가장 높은 tfidf를 가진 피처: \n",
      feature_names[sorted_by_tfidf[-20:]])
```

```
가장 낮은 tfidf를 가진 피처:
 ['provide' 'shooter' 'pushed' 'directly' 'infamous' 'logic' 'fancy'
 'ralli' 'influence' 'decision' 'passionate' 'buddy' 'dreams' 'lying'
 'states' 'deliver' 'eyed' 'path' 'various' 'ad']
가장 높은 tfidf를 가진 피처:
 ['ned' 'europa' 'bad' 'sean' 'book' 'stooges' 'choose' 'showed' '3d'
 'davies' 'game' 'lucy' 'walken' 'blank' 'zombie' 'danish' 'busy' 'xica'
 'documentary' 'master']
```

◦ idf가 낮은 단어 확인

```
sorted_by_idf = np.argsort(vectorizer.idf )
print("가장 낮은 idf를 가진 피처:\n", feature_names[sorted_by_idf[:100]])
```

◦ 예상대로 대부분 'the', 'of' 등과 같은 불용어이다

◦ 'movie', 'film' 등 영화 관련 단어이다

◦ 'good', 'great', 'bad' 등은 감성 분석에서는 중요하겠지만, tf-idf 에서는 덜 중요하다

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

```
가장 낮은 idf를 가진 특성:
 ['the' 'and' 'of' 'to' 'this' 'it' 'is' 'in' 'that' 'but' 'for' 'with'
 'was' 'movie' 'as' 'on' 'not' 'be' 'one' 'film' 'are' 'have' 'you' 'at'
 'all' 'an' 'from' 'so' 'by' 'like' 'who' 'they' 'about' 'just' 'if'
 'there' 'or' 'he' 'his' 'has' 'out' 'some' 'good' 'can' 'what' 'when'
 'more' 'even' 'only' 'story' 'very' 'see' 'up' 'would' 'time' 'my' 'had'
 'no' 'really' 'me' 'her' 'were' 'which' 'made' 'their' 'much' 'also'
 'other' 'been' 'don' 'well' 'because' 'than' 'how' 'do' 'people' 'get'
 'great' 'make' 'will' 'movies' 'first' 'watch' 'could' 'any' 'too' 'she'
 'plot' 'after' 'into' 'most' 'way' 'we' 'bad' 'think' 'seen' 'acting'
 'never' 'its' 'life']
```

# 2.5 모델 계수 조사

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

array([[-0.14979788, -0.88175277, -0.17550195, ..., -0.14843453,
        -0.61978544,  1.30470632]])
```

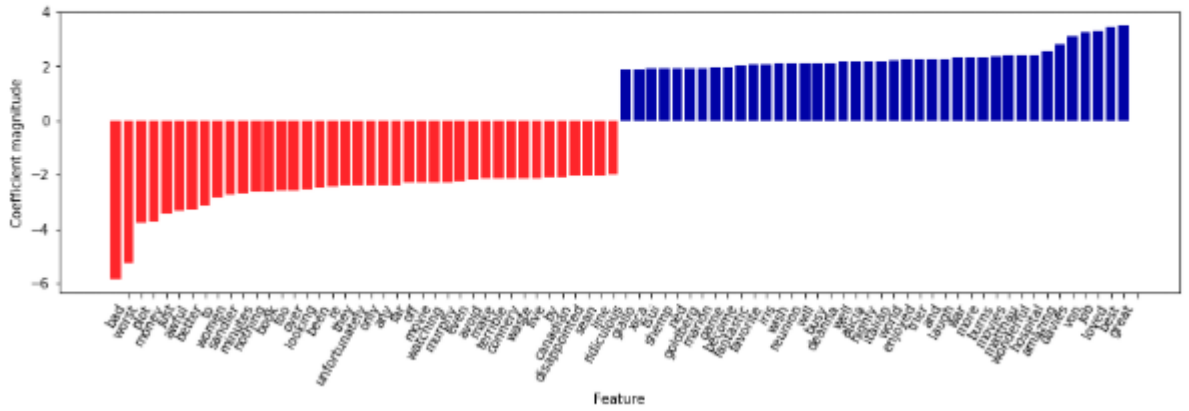
- 로지스틱 회귀 모델

```
grid.best_estimator_.named_steps[“logiticregression”].coef_
```

```
mglearn.tools.visualize_coefficients(
  grid.best_estimator_.named_steps[“logiticregression”].coef_[0],
  feature_names, n_top_features=40
```

  - 왼쪽 음수 계수는 부정적인 리뷰, 오른쪽 양수 계수는 긍정적인 리뷰
  - 왼쪽은 ‘bad’, ‘worst’, ‘plot’ 등이며, 오른쪽은 ‘great’, ‘best’, ‘loved’ 등이다

```
mglearn.tools.visualize_coefficients(
  grid.best_estimator_.named_steps["logisticregression"].coef_[0],
  feature_names, n_top_features=40)
```



# 2.6 여러 단어로 만든 BOW(n-그램)

- BOW 의 약점
  - 단어의 순서를 고려하지 않고 출현 빈도만 계산한다
- 문맥을 고려한 BOW
  - 토큰 하나만 고려하지 않고, 옆에 있는 토큰 두세 개를 함께 고려하는 방법
  - 바이그램 bigram, 트라이그램 trigram, n-그램
  - CountVectorizer와 TfidfVectorizer는 ngram_range 매개변수에 피처의 특성을 고려할 토큰의 범위를 지정할 수 있다
  - ngram_range 매개변수의 입력값은 튜플이며 연속된 토큰의 최소 길이와 최대 길이다

- 샘플 예시

```
print("bards_words:\n", bards_words)
```

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

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("어휘 사전 크기:", len(cv.vocabulary_))
print("어휘 사전:\n", cv.get_feature_names())
```

```
어휘 사전 크기: 13
어휘 사전:
 ['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the', 'think', 'to', 'wise']
```

  - 기본 값은 최소 길이가 1이고 최대 길이가 1인 토큰마다 하나의 피처를 만든다
  - 토큰이 하나이며 토큰 하나를 유니그램 unigram 이라 한다

◦ 토큰 2개가 연속된 바이그램만 만들려면 ngram_range에 (2,2)를 지정한다

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

```
어휘 사전 크기: 14
어휘 사전:
 ['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to', 'is wise', 'knows himself', 'man kn
ows', 'the fool', 'the wise', 'think he', 'to be', 'wise man']
```

◦ 연속된 토큰의 수가 커지면 보통 피처가 더 구체적이고 많이 만들어진다

```
print("변환된 데이터 (밀집 배열):\n", cv.transform(bards_words).toarray())
```

```
변환된 데이터 (밀집 배열):
 [[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

◦ 위의 두 문장 사이에는 공통된 바이그램이 없다.

- 여러 가지 토큰을 고려한 방법

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("어휘 사전 크기:", len(cv.vocabulary_))
print("어휘 사전:\n", cv.get_feature_names())
```

```
어휘 사전 크기: 39
어휘 사전:
 ['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think', 'doth think he', 'fool', 'fool dot
h', 'fool doth think', 'he', 'he is', 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wis
e', 'knows', 'knows himself', 'knows himself to', 'man', 'man knows', 'man knows himself', 'the', 'the fool',
'the fool doth', 'the wise', 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be', 'to be fool',
'wise', 'wise man', 'wise man knows']
```

  - 기본적으로 토큰의 최소 길이는 1이다. 많은 경우에 바이그램을 추가하면 도움이 된다. 5-그램까지는 도움이 되지만, 피처 수가 많아지고 구체적인 피처가 많아지기 때문에 과대적합이 될 수 있다
  - 위는 유니그램, 바이그램, 트라이그램을 적용한 예이다

- IMDb 데이터에 TfidfVectorizer 적용

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
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 배리데이션 점수: {:.2f}”.format(grad.best_score_))
print(“최적의 매개변수:\n”, grid.best_params)
```

  - 조합이 다양하기 때문에 시간이 많이 걸린다(최소 1~2분)

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
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", grid.best_params_)
```

```
C:₩Users₩Win 2₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩logistic.p
r will be changed to 'lbfgs' in 0.22. Specify a solver to silence this war
  FutureWarning)
```

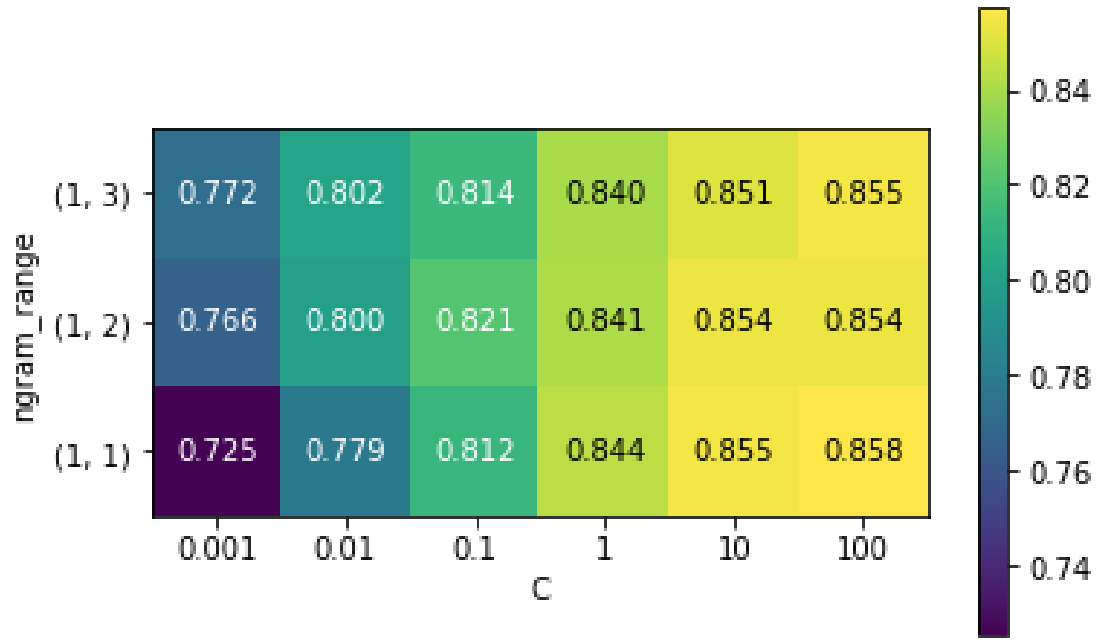
```
최상의 크로스 밸리데이션 점수: 0.86
최적의 매개변수:
 {'logisticregression__C': 100, 'tfidfvectorizer__ngram_range': (1, 1)}
```

- 교차 검증 점수는 86%로 변화가 없었다
- ngram_range가 (1,1)이 되었기 때문에 변화가 없었다고 본다

- 교차 검증 정확도의 히트맵
  - 바이그램, 트라이그램을 사용하면, 정확도가 약간씩 감소한다
  - 데이터 세트를 일부만 사용해서 교재와는 다른 결과가 나왔다

```
# 그리드 서치에서 테스트 점수를 추출합니다
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# 히트맵을 그립니다
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

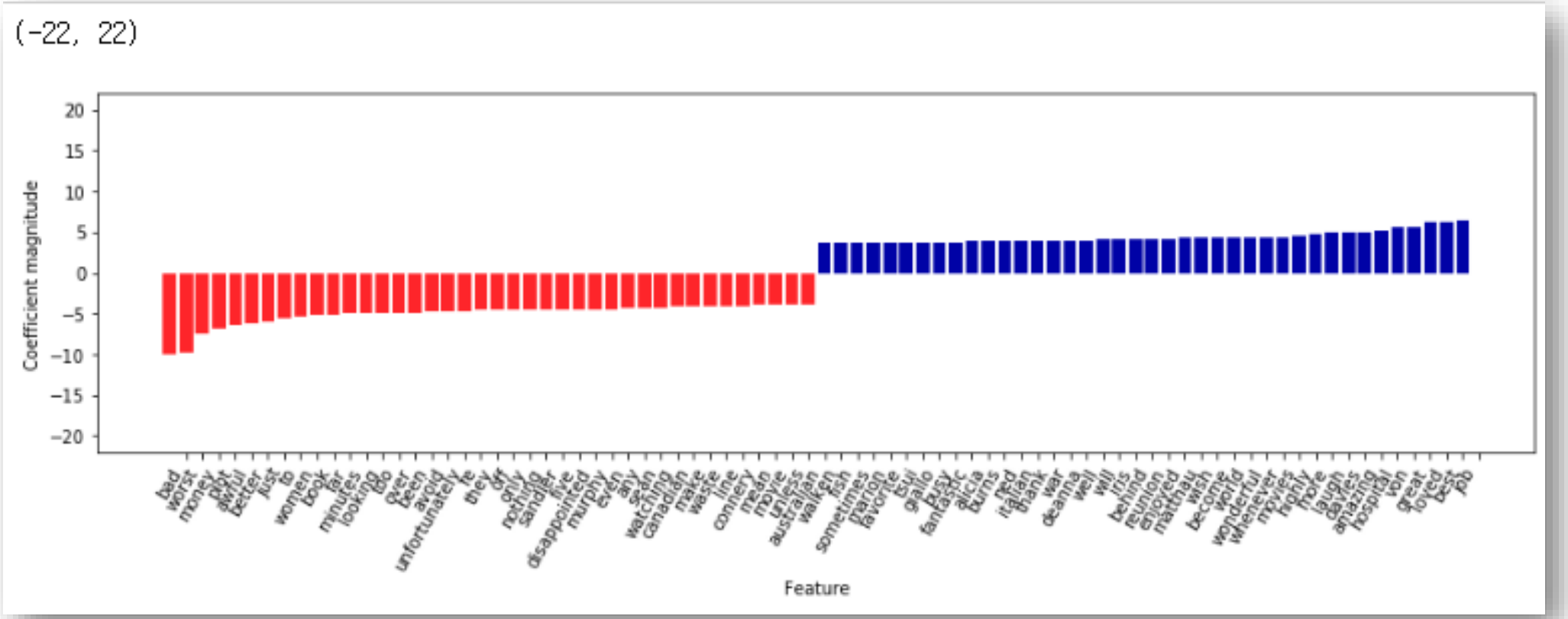
```
<matplotlib.colorbar.Colorbar at 0x24d54574780>
```



- 피처 이름과 계수 추출

```
vect = grid.best_estimator_.named_steps[‘tfidfvectorizer’]
feature_names = np.array(vect.get_feature_names( ))
coef = grid.best_estimator_.named_steps[‘logisticregression’].coef
mglearn.tools.visualize_coefficients(coef[0], feature_names, n_top_features=40)
plt.ylim(-22, 22)
```

```
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef[0], feature_names, n_top_features=40)
plt.ylim(-22, 22)
```




- 앞에서 사용한 로지스틱 회귀와 약간 다른 결과가 나왔다
- 부정적 측면에서 3번째 4번째가 바뀌었고, ‘money’, ‘plot’
- 긍정적인 측면에서 ‘great’ 대신에 ‘job’이 계수가 가장 높았다

# 2.7 고급 토큰화, 어간 추출, 표제어 추출

- 패키지 설치
  - 아나콘다 프롬프트에서

```
pip install –U spacy
pip install nltk
```

  - 쥬피터 노트북에서 영어 모듈을 다운 받는다

```
!python –m spacy download en_core_web_sm
```

```
import spacy
import en_core_web_sm
nlp = en_core_web_sm.load()
import nltk
# 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 = 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])
```

표제어와 어간을 비교하는 함수를 만들었음

◦ 함수를 이용하여 표제어와 어간 비교

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

```
표제어:
['-PRON-', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', '-PRON-', 'be', 'scared', 'of', 'meet',
'the', 'client', 'tomorrow', '.']
어간:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', 'am', 'scare', 'of', 'meet', 'the', 'cl
ient', 'tomorrow', '.']
```

◦ 어간추출은 항상 단어에서 어간만 남겨놓고 제거하므로 ‘was’는 ‘wa’가 된다

◦ 하지만 표제어 추출은 기본 동사형은 ‘be’를 추출했다

◦ 또한 표제어 추출에서 ‘worse’는 ‘bad’로 정규화(통제어휘)시켰지만, 어간추출은 ‘wors’이다

# 요약

- 문자열 데이터 타입 4가지
- 텍스트 데이터를 BOW로 표현하기
  - 토큰화와 어휘사전 구축
  - 단어 추출 방법 변경
    - min_df= 토큰이 나타날 최소 문서 개수 지정
- 불용어 리스트
- 역문헌빈도(tf-idf)
- n-그램 BOW
  - 문맥을 고려한 단어 출현 빈도
- 어간 추출과 표제어 추출