

1 데이터분석 단계(Data Analysis Cycle)

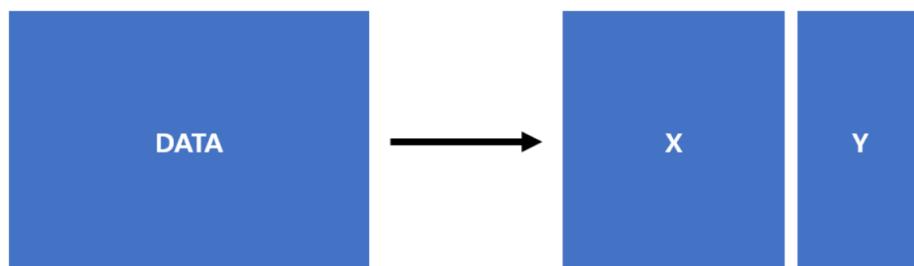
✓ 데이터 전처리: (0) 쓸모 없을 뻔한 Raw를 쓸모 있는 Data로 변환

	100	T50	횟수	111	TPU	...
few	Gds	Hvi	Rew	Fa	...	
Fre	CT	QTP	D	합	...	
'1'	1	23	22	NaN	43	
76	NaN	43	32	1	8	
'Hi'	NaN	NaN	NaN	NaN	87	
23	98	NaN	64	46	NaN	
c	90	NaN	'WW'	24	'KK'	
t	NaN	2	NaN	NaN	6	
64	NaN	90	'IU'	4	76	

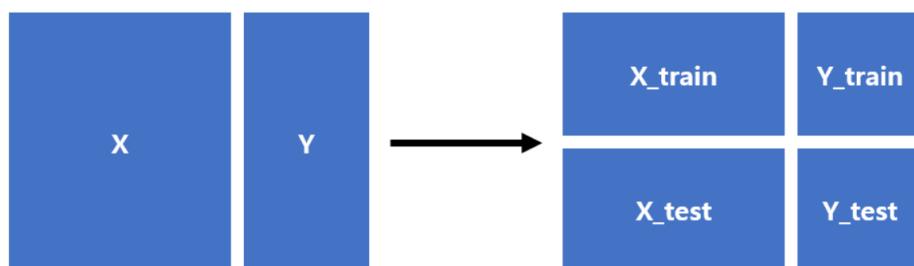
→

번호	시간	총량	기간	누적	...
1	1	23	22	21	43
76	33.3	43	32	1	8
5	33.3	52	35	21	87
23	98	52	64	46	61
90	33.3	2	24	33	4
55	2	52	35	21	6
64	33.3	90	11	4	76

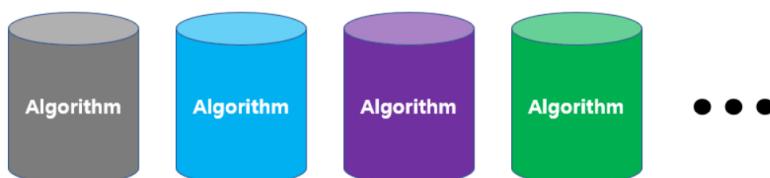
✓ 데이터 분할: (1) 목표/종속변수 Y와 설명/독립변수 X설정



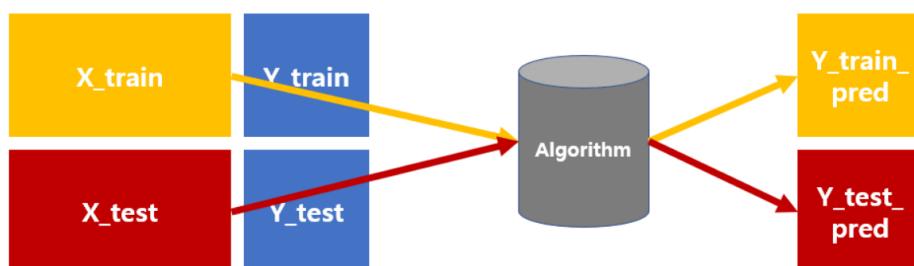
✓ 데이터 분할: (2) 학습데이터 Train과 예측 데이터 Test로 분할



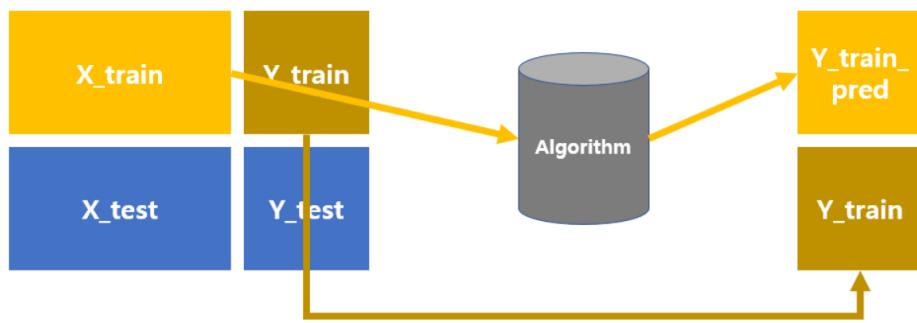
✓ 모델링: (3) 분석 목적에 맞는 알고리즘(Base & Advanced) 후보들 준비



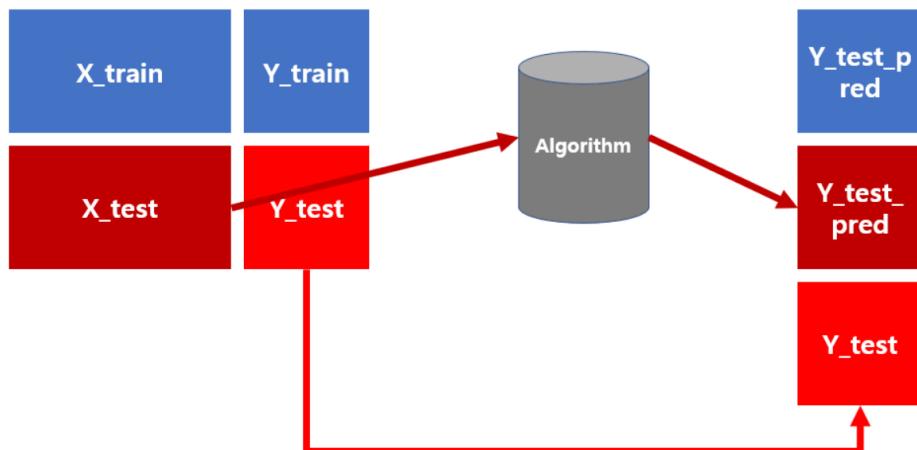
✓ 모델링 & 학습: (4) 알고리즘 평가를 위해 Train/Test의 예측값 추정



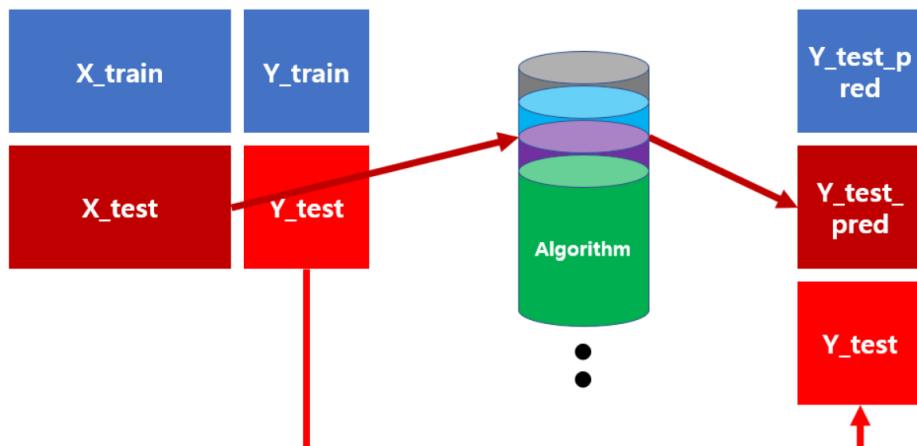
✓ 평가: (5) 학습(Train)이 잘 되었는지 알고리즘 성능검증



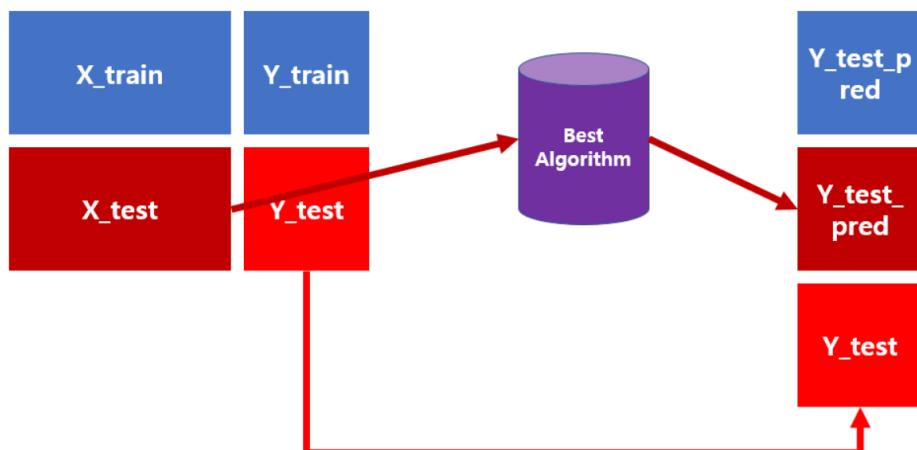
✓ 평가: (6) 예측(Test)이 잘 되었는지 알고리즘 성능검증



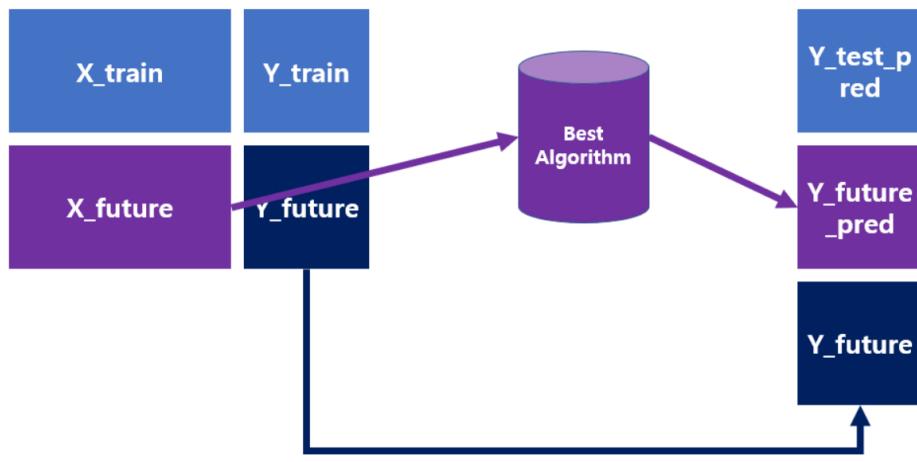
✓ 최적 알고리즘 선택: (7) 알고리즘을 변경하여 위 과정 반복 후



✓ 최적 알고리즘 선택: (7) 최고 성능의 알고리즘 선택



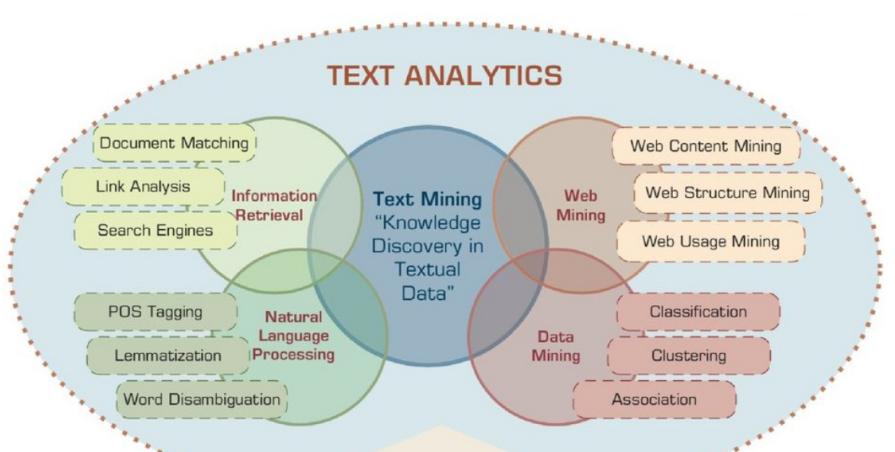
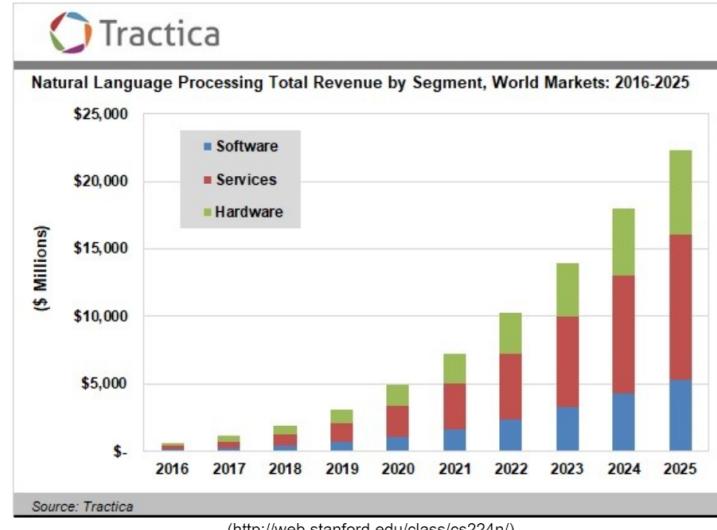
✓ 배포: (8) 실제 비즈니스 서비스 현업 적용 및 매출/수익/개선 정도 평가

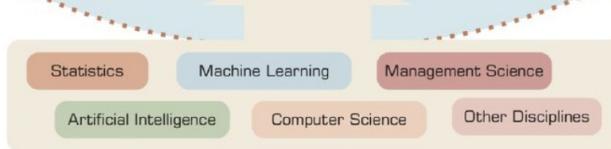


2 텍스트마이닝(Text Mining)

1) **Text Analytics:** 컴퓨터와 인간언어의 상호작용 분야에서 출발했으며, 기계가 인간의 언어를 이해하고 해석하기 위해 텍스트에서 의미 있는 정보를 추출하는 기술

	데이터마이닝	텍스트마이닝
데이터구조	정형/구조화 데이터	비정형/비구조화 데이터
데이터종류	수치형 또는 범주형 데이터	문자열 데이터
데이터표현	직관적	복합적
데이터차원	1만차원 이하	1만차원 이상
데이터전처리	변수선택	변수추출
방법론	통계추론/기계학습/딥러닝 등	텍스트정보추출 + 통계추론/기계학습/딥러닝 등
활용시장	중소대 기업 모두 + 거시(집단)레벨 분석 필요시	주로 대기업 + 미시(개인)레벨 분석 필요시
확장시점	약 1994년 후	약 2000년 후

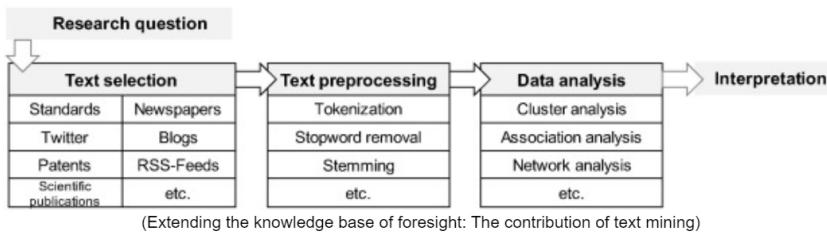




Source: Ramesh Sharda, Dursun Delen, and Elraim Turban (2017), *Business Intelligence, Analytics, and Data Science: A Managerial Perspective*, 4th Edition, Pearson

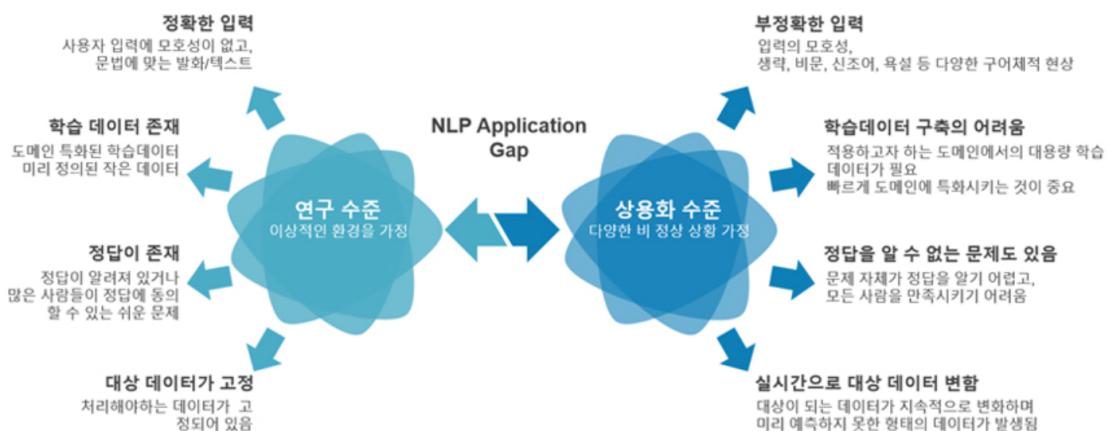
2) 텍스트마이닝(Text Mining): 인간의 언어는 다양한 구조, 체계, 규칙, 문법 등 복잡한 요소들로 구성되어 있고, 자연어 처리는 이러한 요소들을 컴퓨터에게 전달해 인간의 언어를 이해 할 수 있도록 패턴을 추출하는 등의 방법을 통해 분석 및 처리 하는 방법

- (1) 웹/소셜/블로그 등으로부터 확득한 데이터를
- (2) 자연어 처리 과정을 거쳐 비정형 데이터 → 정형 데이터로 변경하고
 - 형태소 분석(Morphological Analysis): 문장을 최소의 단위로 분리하는 과정
- (3) 분석 목적과 용도에 맞는 알고리즘이 인식 할 수 있게 변환한 후
- (4) 텍스트 마이닝 알고리즘을 통해 발견한 규칙과 패턴을 해석 및 평가 하여
 - 구문 분석(Syntax Analysis): 문장의 구조와 형태를 파악하는 단계로 언어의 문법을 이용하여 구조 추출
 - 의미 분석(Semantic Analysis): 구문 분석 결과에 해석을 더하여 문장이 가진 의미 파악하는 과정
 - 화용 분석(Pragmatic Analysis): 문장의 숨은 의미나 대화의 내용 이해하는 과정
- (5) 최종적으로 데이터를 자식화 하는 과정
 - 자연어의 정확한 이해가 필요한 인공지능 응용분야(검색엔진/챗봇/번역)의 경우 화용분석까지 활용하여 자식화
 - 일반적으로 비구조화 데이터를 구조화 데이터로 변경하여 통계/데이터마이닝을 이용한 자식화는 형태소 분석만 활용



3) 텍스트마이닝 기법:

분석 기법	설명
텍스트 요약(Summarization)	텍스트 내 의미있는 주요 주제나 중심 사상을 추출하는 토픽 모델링(Topic Modeling)에 활용
텍스트 예측(Text Prediction)	기존의 문서나 말의 내용을 기반으로 향후 등장할 단어나 문장을 예측 지도학습 방식으로 주로 학습하여 구글번역, 이미지 자동완성, 챗봇 등에 활용
텍스트 분류(Text Classification)	문서가 특정 분류 또는 카테고리에 속하는지 예측 지도 학습 방식으로 주로 학습하여 기사 카테고리 분류 및 스팸 메일 검출 등에 활용
텍스트 군집화(Clustering)	비슷한 유형의 문서를 모아 유사도 기반 군집화를 수행 빈번하게 출현하는 유의미한 동시 출현 단어쌍들을 네트워크 분석으로 관계 파악
감정 분석(Sentiment Analysis)	텍스트에서 나타나는 의견이나 감정을 분석하는 기법 소셜 미디어 감정 분석, 영화나 제품의 리뷰 긍부정 분석, 사회 이슈파악, 여론분석이나 심리분석 등 텍스트 분석에서 가장 활발하게 활용



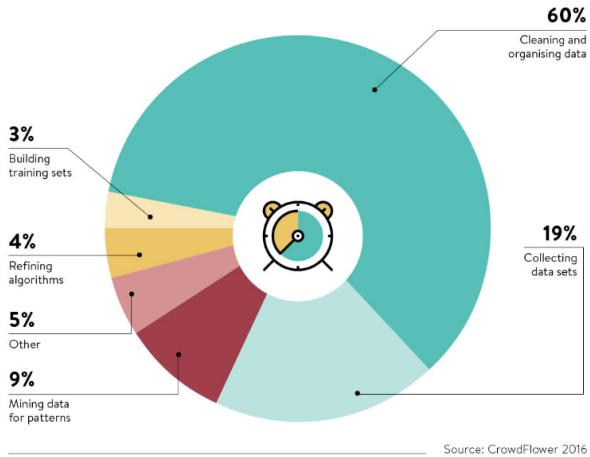
(커뮤니케이션과 AI #6 자연어 처리 기술의 다양한 상황들 by 엔씨소프트의 NLP(자연어처리) 센터)

3 전처리 방향(Preprocessing)

• 목표:

- 대량으로 수집된 데이터는 그대로 활용 어려움
- 잘못 수집/처리 된 데이터는 엉뚱한 결과를 발생

WHAT DATA SCIENTISTS SPEND THE MOST TIME DOING



일반적인 전처리 필요항목:

- 데이터 결합
- 결측값 처리
- 이상치 처리
- 자료형 변환
- 데이터 분리
- 데이터 변환
- 스케일 조정

⇒ "알고리즘의 범위와 종류가 다양하여, 각 알고리즘의 입력에 맞게 변환 하는 것이 최선"

⇒ "횡단면 데이터 및 시계열 데이터가 혼합된 패널데이터의 특성이 많기 때문에 일반 전처리와 시계열 전처리 그리고 데이터 차원증가와 감소 등 다양한 전처리 도구와 능력이 필요하고 다양하고 자유로운 변환이 필요한 고차원 전처리 능력 필요"

4 모델링 방향(Modeling)

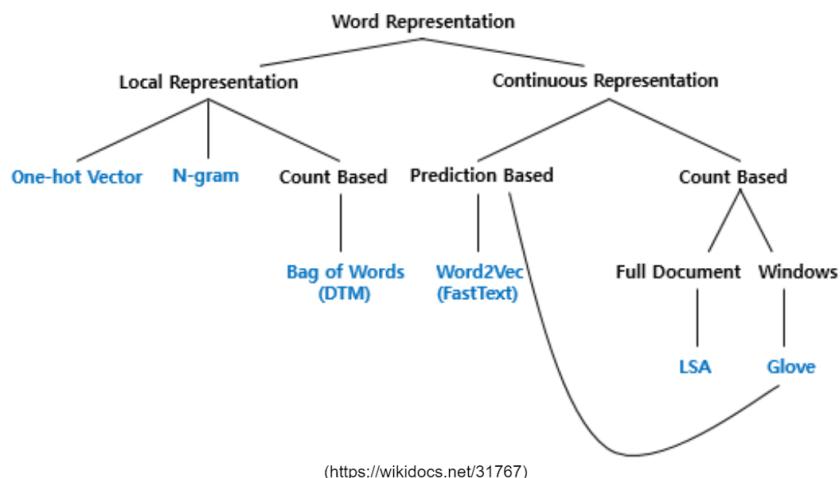
"단어의 표현 방법은 크게 국소표현(Local Representation) 및 분산표현(Distributed Representation) 방식으로 개발"

1) 국소표현: 각 단어마다 특성 값을 매칭 하여 단어를 표현하는 것으로, 이산표현(Discrete Representation)이라고도 함

- puppy, cute, lovely라는 단어가 있으면 차례대로 1, 2, 3을 매핑
- 각 단어들의 관련성이나 뉘앙스를 표현하기 어려움

2) 분산표현: 단어를 표현하기 위해 주변의 단어를 참고하여 추론 하는 것으로, 연속표현(Continuous Representation)이라고도 함

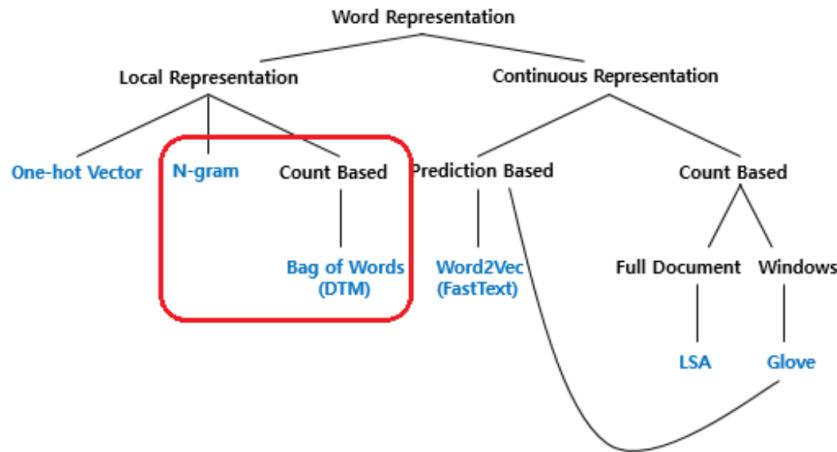
- puppy 근처에 cute, lovely가 자주 등장하기 때문에, cute + lovely 느낌이라고 단어를 추론
- 각 단어들의 관련성이나 뉘앙스를 표현 가능



4.1 빈도기반 단어표현

"빈번하게 출현하는 단어들을 기반으로 문서나 단어를 군집화 표현하여 네트워크 분석을 하기도 함"

- One-Hot Encoding
- Bag of Words(BoW)
- Document-Term Matrix(DTM)
- Bag of N-gram(BoN)
- TF-IDF



4.1.1 Bag of Words(BoW)

"단어들의 순서는 전혀 고려치 않고 단어들의 출현 빈도에만 집중 하는 텍스트 수치 벡터화"

- (1) 단어집합 내 각 단어에 고유한 정수 인덱스 부여
- (2) 각 인덱스에 단어의 빈도를 기록한 벡터 생성

예시	정부가 발표하는 물가상승률과 소비자가 느끼는 물가상승률은 다르다
형태소 인덱싱	{'정부': 0, '가': 1, '발표': 2, '하는': 3, '물가상승률': 4, '파': 5, '소비자': 6, '느끼는': 7, '은': 8, '다르다': 9}
BoW 벡터화	[1, 2, 1, 1, 2, 1, 1, 1, 1, 1]

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

```
In [1]: # 예시 텍스트 데이터
sentence = ["정부가 발표하는 물가상승률과 소비자가 느끼는 물가상승률은 다르다."]
# 텍스트 데이터를 단어별 빈도로 정수 인코딩
from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer()
tokenizer.fit(sentence)
print(tokenizer.vocabulary_)
print(tokenizer.transform(sentence).toarray())
{'정부': 6, '발표하는': 4, '물가상승률과': 2, '소비자가': 5, '느끼는': 0, '물가상승률은': 3, '다르다': 1}
[[1 1 1 1 1 1]]
```

```
In [2]: # 예시 텍스트 데이터
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords

sentences = "A barber is a person. a barber is good person. #
           a barber is huge person. he Knew A Secret! The Secret He Kept is huge secret. #
           Huge secret. His barber kept his word. a barber kept his word. #
           His barber kept his secret. #
           But keeping and keeping such a huge secret to himself was driving the barber crazy. #
           the barber went up a huge mountain."
```

```
# 단어 토큰화 및 빈도수 추정
stop_words = set(stopwords.words('english'))

sentence_pred = []
vocab = dict()
for sentence in sent_tokenize(sentences):
    # 단어 토큰화
    tokenized_sentence = word_tokenize(sentence)
    result = []
    for word in tokenized_sentence:
        # 모든 단어의 소문자화
        word = word.lower()
        # 불용어 제거
        if word not in stop_words:
            # 단어 길이가 2이하인 제거
            if len(word) > 2:
                result.append(word)
                # 결과 단어 정리 및 빈도수 계산
                if word not in vocab:
                    vocab[word] = 0
                    vocab[word] = vocab[word] + 1
    sentence_pred.append(result)
```

```

print(sentence_pred, '#\n#\n', vocab)

[['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], ['keeping', 'keepin
g', 'huge', 'secret', 'driving', 'barber', 'crazy'], ['barber', 'went', 'huge', 'mountain']]

{'barber': 8, 'person': 3, 'good': 1, 'huge': 5, 'knew': 1, 'secret': 6, 'kept': 4, 'word': 2, 'keeping': 2, 'driving': 1, 'crazy': 1, 'went': 1, 'mountain': 1}

In [3]: # 정제된 텍스트를 재문장화
sentence_prep = " ".join(i) for i in sentence_pred
sentence_prep

```

```

Out[3]: ['barber person',
'barber good person',
'barber huge person',
'knew secret',
'secret kept huge secret',
'huge secret',
'barber kept word',
'barber kept word',
'barber kept secret',
'keeping keeping huge secret driving barber crazy',
'barber went huge mountain']

```

```

In [4]: # 원 텍스트 데이터를 단어별 빈도로 정수 인코딩화
from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer(max_features=5)
sentence_encoded = []
for each in sentence_prep:
    sentence_freq = tokenizer.fit_transform([each])
    sentence_encoded.append(list(sentence_freq.toarray().ravel()))
sentence_encoded

```

```

Out[4]: [[1, 1],
[1, 1, 1],
[1, 1, 1],
[1, 1],
[1, 1, 2],
[1, 1],
[1, 1, 1],
[1, 1, 1],
[1, 1, 1],
[1, 1, 1, 1, 2],
[1, 1, 1, 1]]

```

```

In [5]: # 최대길이에 맞춰 제로 패딩
# 기계학습 알고리즘에선 의미없는 0단어는 무시할 것
from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer(max_features=10)
tokenizer.fit(sentence_prep)
sentence_encoded = tokenizer.transform(sentence_prep).toarray()
sentence_encoded = sentence_encoded.tolist()
sentence_encoded

```

```

Out[5]: [[1, 0, 0, 0, 0, 0, 0, 1, 0, 0],
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0],
[1, 0, 0, 0, 1, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 1, 0, 1, 0, 2, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
[1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
[1, 0, 0, 0, 0, 0, 1, 0, 1, 0],
[1, 1, 1, 0, 1, 2, 0, 0, 1, 0],
[1, 0, 0, 0, 1, 0, 0, 0, 0, 0]
]

```

2) 문서 단어 행렬(Document-Term Matrix, DTM):

- 각 문서의 BoW를 사용하여, 서로 다른 문서들의 BoW를 결합한 표현 방식

```

In [6]: # DTM 행렬 표기
import numpy as np
import pandas as pd

colnames = [key for key, value in sorted(tokenizer.vocabulary_.items(), key=lambda item: item[1])]
idxnames = ['sentence' + str(i+1) for i in range(np.shape(sentence_encoded)[0])]
DTM = pd.DataFrame(sentence_encoded, columns=colnames, index=idxnames)
DTM

```

	barber	crazy	driving	good	huge	keeping	kept	person	secret	word
sentence1	1	0	0	0	0	0	0	1	0	0
sentence2	1	0	0	1	0	0	0	1	0	0
sentence3	1	0	0	0	1	0	0	1	0	0
sentence4	0	0	0	0	0	0	0	0	1	0
sentence5	0	0	0	0	1	0	1	0	2	0
sentence6	0	0	0	0	1	0	0	0	1	0
sentence7	1	0	0	0	0	0	1	0	0	1
sentence8	1	0	0	0	0	0	1	0	0	1

sentence9	1	0	0	0	0	0	1	0	1	0
sentence10	1	1	1	0	1	2	0	0	1	0
sentence11	1	0	0	0	1	0	0	0	0	0

```
In [7]: # DTM을 회전하여 각 단어별 문장/문서에 등장횟수로 문장/문서의 주제와 상관 추론 가능
# TDM: Term Document Matrix
TDM = pd.DataFrame(DTM.T.sum(axis=1), columns=['sentence_count'])
TDM
```

```
Out[7]:    sentence_count
barber           8
crazy            1
driving          1
good             1
huge             5
keeping          2
kept              4
person           3
secret            6
word              2
```

- 문서 또는 문장들의 주요단어나 주제어를 수치화하여 비교할 수 있는 장점
- 빈도수를 기반으로만 판단하기 때문에, 중요단어와 불필요 단어 모두 빈도수 높을 수도
- DTM도 원-핫 인코딩의 단점을 포함하기에, 단어의 갯수가 방대해지면 데이터 차원도 함께 증가하여 많은 양의 저장공간과 컴퓨팅 리소스를 필요

4.1.2 Bag of N-gram(BoN)

- DTM은 문맥상 의미를 무시하기 때문에 이를 보완하기 위해 N개 단어를 묶어 Feature 구성

```
In [8]: # 원 텍스트 데이터를 단어별 빈도로 정수 인코딩화
# ngram_range=(1,2): 토큰화된 단어를 1개씩 순서대로 2개씩 묶어서 Feature 구성
from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer(max_features=5, ngram_range=(1,1))
sentence_encoded = []
for each in sentence_prep:
    print(each)
    sentence_freq = tokenizer.fit_transform([each])
    print(tokenizer.get_feature_names_out(), list(sentence_freq.toarray().ravel()))
    sentence_encoded.append(list(sentence_freq.toarray().ravel()))

barber person
['barber' 'person'] [1, 1]
barber good person
['barber' 'good' 'person'] [1, 1, 1]
barber huge person
['barber' 'huge' 'person'] [1, 1, 1]
knew secret
['knew' 'secret'] [1, 1]
secret kept huge secret
['huge' 'kept' 'secret'] [1, 1, 2]
huge secret
['huge' 'secret'] [1, 1]
barber kept word
['barber' 'kept' 'word'] [1, 1, 1]
barber kept word
['barber' 'kept' 'word'] [1, 1, 1]
barber kept secret
['barber' 'kept' 'secret'] [1, 1, 1]
keeping keeping huge secret driving barber crazy
['barber' 'crazy' 'driving' 'huge' 'keeping'] [1, 1, 1, 1, 2]
barber went huge mountain
['barber' 'huge' 'mountain' 'went'] [1, 1, 1, 1]
```

```
In [9]: # 원 텍스트 데이터를 단어별 빈도로 정수 인코딩화
# ngram_range=(1,2): 토큰화된 단어를 1개씩 순서대로 2개씩 묶어서 Feature 구성
from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer(max_features=5, ngram_range=(1,3))
sentence_encoded = []
for each in sentence_prep:
    print(each)
    sentence_freq = tokenizer.fit_transform([each])
    print(tokenizer.get_feature_names_out(), list(sentence_freq.toarray().ravel()))
    sentence_encoded.append(list(sentence_freq.toarray().ravel()))

barber person
['barber' 'barber person' 'person'] [1, 1, 1]
barber good person
['barber' 'barber good' 'barber good person' 'good' 'good person'] [1, 1, 1, 1, 1]
barber huge person
['barber' 'barber huge' 'barber huge person' 'huge' 'huge person'] [1, 1, 1, 1, 1]
knew secret
['knew' 'knew secret' 'secret'] [1, 1, 1]
secret kept huge secret
['huge' 'huge secret' 'kept' 'kept huge' 'secret'] [1, 1, 1, 1, 2]
huge secret
```

```

['huge' 'huge secret' 'secret'] [1, 1, 1]
barber kept word
['barber' 'barber kept' 'barber kept word' 'kept' 'kept word'] [1, 1, 1, 1, 1]
barber kept word
['barber' 'barber kept' 'barber kept word' 'kept' 'kept word'] [1, 1, 1, 1, 1]
barber kept secret
['barber' 'barber kept' 'barber kept secret' 'kept' 'kept secret'] [1, 1, 1, 1, 1]
keeping keeping huge secret driving barber crazy
['barber' 'keeping' 'keeping' 'keeping keeping huge' 'secret'] [1, 2, 1, 1, 1]
barber went huge mountain
['barber' 'barber went' 'barber went huge' 'huge' 'huge mountain'] [1, 1, 1, 1, 1]

```

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

- **BoW:** 빈도수를 기반으로만 판단하기 때문에, 중요단어와 불필요 단어 모두 빈도수 높을 수도

방법론	방향
One-Hot Encoding, BoW, BoN	텍스트 안의 모든 단어가 동일하게 중요하다 가정
TF-IDF	텍스트 안의 특정 단어가 다른 단어 대비 상대적 중요도 가정

- 중요단어에 가중치를 높이고 불필요단어에 가중치를 낮추는 방향 모색
- 단어의 빈도와 역문서빈도를 결합하여 DTM 내 각 단어들마다 중요점도를 가중치로 부여
- 문서의 중요도 또는 유사도 추정, 검색 결과 중요도 추정 등의 특정 정보 추출이나 텍스트 분류에 주로 사용

(1) **TF = DTM:** 특정 문서에서 총 단어수 대비 특정 단어 등장 횟수

(2) **DF = TDM:** 특정 단어가 등장한 문서의 수

(3) **IDF:** DF에 반비례하는 수

$$IDF = \log\left(\frac{n}{1 + DF}\right) = \log\left(\frac{\text{총문서의수}}{\text{특정단어를포함하는문서의수}}\right)$$

- 분모가 0인 상황 방지하기 위해 1을 더하고, IDF값의 상한선을 정하기 위해 log 사용

(4) **TF-IDF:** TF × IDF

비교대상	방향
TF	단어의 중요도는 문서 내 빈도에 비례하여 증가
IDF	단어의 중요도는 다른 문서 내 빈도에 비례하여 감소

- IDF는 TF에 가중치로 반영되는 조절값으로 여러 문서에 나오면 중요도를 낮추는 기능
- 대중적인 단어는 많이 언급되어 중요값, 대중적이지 않은 단어는 덜 언급되어도 중요값
- 중요도 낮음: 모든 문서에서 빈도가 높은 경우에 낮은 수치 (ex. 불용어)
- 중요도 높음: 특정 문서에서만 빈도가 높은 경우에 높은 수치

불용어

일반적으로 CountVectorizer 보다 좋은 결과를 보이는 편
from sklearn.feature_extraction.text import TfidfVectorizer

```

In [10]: # 예시 텍스트 데이터
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords

sentences = "A barber is a person. a barber is good person. #
             a barber is huge person. he Knew A Secret! The Secret He Kept is huge secret. #
             Huge secret. His barber kept his word. a barber kept his word. #
             His barber kept his secret. #
             But keeping and keeping such a huge secret to himself was driving the barber crazy. #
             the barber went up a huge mountain." #

# 단어 토큰화 및 빈도수 추정
stop_words = set(stopwords.words('english'))

sentence_pred = []
vocab = dict()
for sentence in sent_tokenize(sentences):
    # 단어 토큰화
    tokenized_sentence = word_tokenize(sentence)
    result = []
    for word in tokenized_sentence:
        # 모든 단어의 소문자화
        word = word.lower()
        # 불용어 제거
        if word not in stop_words:
            # 단어 길이가 2이하는 제거
            if len(word) > 2:
                result.append(word)
                # 결과 단어 정리 및 빈도수 계산
                if word not in vocab:
                    vocab[word] = 0
                    vocab[word] = vocab[word] + 1
    sentence_pred.append(result)

print(sentence_pred, '#n#n', vocab)

```

```

[['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['kept', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], ['keeping', 'keepin'], ['huge', 'secret', 'driving'], ['barber', 'crazy'], ['barber', 'went', 'huge', 'mountain']]

```

```
{'barber': 8, 'person': 3, 'good': 1, 'huge': 5, 'knew': 1, 'secret': 6, 'kept': 4, 'word': 2, 'keeping': 2, 'driving': 1, 'crazy': 1, 'went': 1, 'mountain': 1}
```

```
In [11]: # 정제완료 토큰들의 재문장화  
sentence_prep = " ".join(i) for i in sentence_pred]  
sentence_prep
```

```
Out[11]: ['barber person',  
          'barber good person',  
          'barber huge person',  
          'knew secret',  
          'secret kept huge secret',  
          'huge secret',  
          'barber kept word',  
          'barber kept word',  
          'barber kept secret',  
          'keeping keeping huge secret driving barber crazy',  
          'barber went huge mountain']
```

```
In [12]: # 최대길이에 맞춰 새로 패딩  
# 기계학습 알고리즘에선 의미없는 0단어는 무시할 것  
from sklearn.feature_extraction.text import CountVectorizer  
  
tokenizer = CountVectorizer(max_features=10)  
tokenizer.fit(sentence_prep)  
sentence_encoded = tokenizer.transform(sentence_prep).toarray()  
sentence_encoded = sentence_encoded.tolist()  
sentence_encoded
```

```
Out[12]: [[1, 0, 0, 0, 0, 0, 0, 1, 0, 0],  
          [1, 0, 0, 1, 0, 0, 0, 1, 0, 0],  
          [1, 0, 0, 0, 1, 0, 0, 1, 0, 0],  
          [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  
          [0, 0, 0, 0, 1, 0, 1, 0, 2, 0],  
          [0, 0, 0, 0, 1, 0, 0, 0, 1, 0],  
          [1, 0, 0, 0, 0, 0, 1, 0, 0, 1],  
          [1, 0, 0, 0, 0, 0, 1, 0, 0, 1],  
          [1, 0, 0, 0, 0, 0, 1, 0, 1, 0],  
          [1, 1, 1, 0, 1, 2, 0, 0, 1, 0],  
          [1, 0, 0, 0, 1, 0, 0, 0, 0, 0]]
```

```
In [13]: # 수동으로 TF-IDF 계산  
# DTM 행렬 보기  
import pandas as pd  
  
colnames = [key for key, value in sorted(tokenizer.vocabulary_.items(), key=lambda item: item[1])]  
idxnames = ['sentence' + str(i+1) for i in range(np.shape(sentence_encoded)[0])]  
DTM = pd.DataFrame(sentence_encoded, columns=colnames, index=idxnames)  
  
# DTM을 회전하여 각 단어별 문장/문서에 등장횟수로 문장/문서의 주제와 사상 추론 가능  
# TDM: Term Document Matrix  
TDM = pd.DataFrame(DTM.T.astype(bool).sum(axis=1), columns=['sentence_count'])  
  
# IDF 추정  
IDF = np.log(DTM.shape[0] / (TDM + 1))  
display(DTM, IDF)  
  
# TF-IDF 추정  
TF_IDF = pd.DataFrame(DTM.values * IDF.values.T, columns=DTM.columns, index=DTM.index)  
TF_IDF
```

	barber	crazy	driving	good	huge	keeping	kept	person	secret	word
sentence1	1	0	0	0	0	0	0	1	0	0
sentence2	1	0	0	1	0	0	0	1	0	0
sentence3	1	0	0	0	1	0	0	1	0	0
sentence4	0	0	0	0	0	0	0	0	1	0
sentence5	0	0	0	0	1	0	1	0	2	0
sentence6	0	0	0	0	1	0	0	0	1	0
sentence7	1	0	0	0	0	0	1	0	0	1
sentence8	1	0	0	0	0	0	1	0	0	1
sentence9	1	0	0	0	0	0	1	0	1	0
sentence10	1	1	1	0	1	2	0	0	1	0
sentence11	1	0	0	0	1	0	0	0	0	0

	sentence_count
barber	0.200671
crazy	1.704748
driving	1.704748
good	1.704748
huge	0.606136
keeping	1.704748
kept	0.788457
person	1.011601
secret	0.606136
word	1.299283

Out [13] :

	barber	crazy	driving	good	huge	keeping	kept	person	secret	word
sentence1	0.200671	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.011601	0.000000	0.000000
sentence2	0.200671	0.000000	0.000000	1.704748	0.000000	0.000000	0.000000	1.011601	0.000000	0.000000
sentence3	0.200671	0.000000	0.000000	0.000000	0.606136	0.000000	0.000000	1.011601	0.000000	0.000000
sentence4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.606136	0.000000
sentence5	0.000000	0.000000	0.000000	0.000000	0.606136	0.000000	0.788457	0.000000	1.212272	0.000000
sentence6	0.000000	0.000000	0.000000	0.000000	0.606136	0.000000	0.000000	0.000000	0.606136	0.000000
sentence7	0.200671	0.000000	0.000000	0.000000	0.000000	0.000000	0.788457	0.000000	0.000000	1.299283
sentence8	0.200671	0.000000	0.000000	0.000000	0.000000	0.000000	0.788457	0.000000	0.000000	1.299283
sentence9	0.200671	0.000000	0.000000	0.000000	0.000000	0.000000	0.788457	0.000000	0.606136	0.000000
sentence10	0.200671	1.704748	1.704748	0.000000	0.606136	3.409496	0.000000	0.000000	0.606136	0.000000
sentence11	0.200671	0.000000	0.000000	0.000000	0.606136	0.000000	0.000000	0.000000	0.000000	0.000000

In [14] :

```
# sklearn 사용
# Scikit-Learn의 IDF 공식 차이:  $IDF(t) = \log((1+n) / (1+df(t))) + 1$ 
# Scikit-Learn의 TF-IDF 결과를 유clidean 거리(L2 normalize)를 적용
from sklearn.feature_extraction.text import TfidfVectorizer

tfidifier = TfidfVectorizer(max_features=10)
tfidifier.fit(sentence_prep)
print(sorted(tfidifier.vocabulary_.items()))
TF_IDF_sklearn = tfidifier.transform(sentence_prep).toarray()
TF_IDF_sklearn = pd.DataFrame(TF_IDF_sklearn,
                               columns=tfidifier.get_feature_names_out(),
                               index=DTM.index)

TF_IDF_sklearn
[('barber', 0), ('crazy', 1), ('driving', 2), ('good', 3), ('huge', 4), ('keeping', 5), ('kept', 6), ('person', 7), ('secret', 8), ('word', 9)]
```

Out [14] :

	barber	crazy	driving	good	huge	keeping	kept	person	secret	word
sentence1	0.522986	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.852341	0.000000	0.000000
sentence2	0.345928	0.000000	0.000000	0.749999	0.000000	0.000000	0.000000	0.563780	0.000000	0.000000
sentence3	0.430929	0.000000	0.000000	0.000000	0.5666620	0.000000	0.000000	0.702311	0.000000	0.000000
sentence4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
sentence5	0.000000	0.000000	0.000000	0.000000	0.400739	0.000000	0.443892	0.000000	0.801479	0.000000
sentence6	0.000000	0.000000	0.000000	0.000000	0.0707107	0.000000	0.000000	0.000000	0.707107	0.000000
sentence7	0.390567	0.000000	0.000000	0.000000	0.000000	0.000000	0.568849	0.000000	0.000000	0.723787
sentence8	0.390567	0.000000	0.000000	0.000000	0.000000	0.000000	0.568849	0.000000	0.000000	0.723787
sentence9	0.454067	0.000000	0.000000	0.000000	0.000000	0.000000	0.661334	0.000000	0.597043	0.000000
sentence10	0.174980	0.379366	0.379366	0.000000	0.230078	0.758732	0.000000	0.000000	0.230078	0.000000
sentence11	0.605349	0.000000	0.000000	0.000000	0.795961	0.000000	0.000000	0.000000	0.000000	0.000000

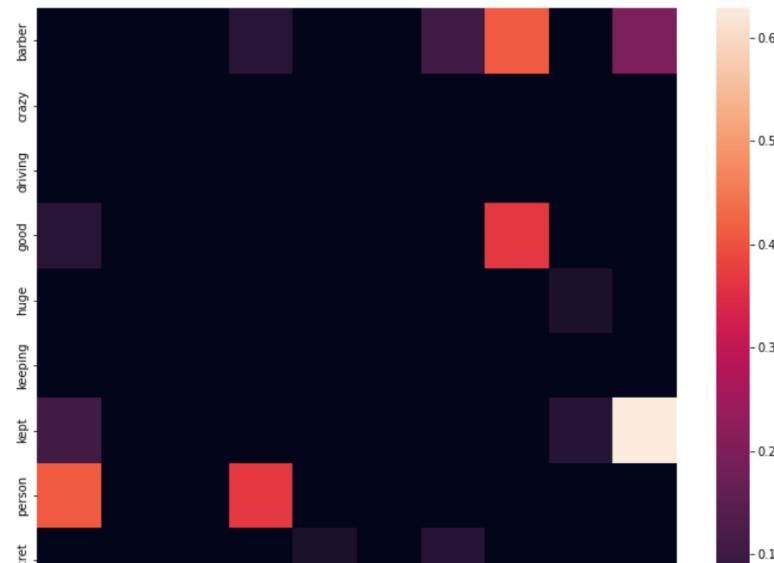
In [17] :

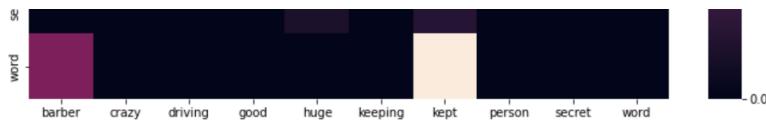
```
# TF-IDF 상관성 확인을 통한 동시 출현 단어 분석
TFIDF_corr = TF_IDF_sklearn.corr()
```

```
# 대각행 결과 음수관계 삭제
TFIDF_corr[TFIDF_corr == 1] = 0
TFIDF_corr[TFIDF_corr < 0] = 0
```

```
# 시각화
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(12,10))
sns.heatmap(TFIDF_corr)
plt.show()
```





```
In [18]: # Train 데이터로 TF-IDF 예측
tfidfier.transform(['barber person']).toarray()

Out[18]: array([[0.52298591, 0.          , 0.          , 0.          , 0.          ,
   0.          , 0.85234133, 0.          , 0.          , 0.]])
```

```
In [19]: # Test 데이터로 TF-IDF 예측
tfidfier.transform(['barber and person are keeping driving']).toarray()

Out[19]: array([[0.27674405, 0.          , 0.59999501, 0.          , 0.          ,
   0.59999501, 0.          , 0.45102628, 0.          , 0.]])
```

- 몇 가지 단점:

- 단어가 나온 위치나 앞뒤관계, 문맥, 문서상의 순서 등은 고려하지 못함
- 순수하게 횟수에만 의존하기 때문에 한국어에는 좋지 않은 성능이 될 수
- 기계학습/머신러닝/딥러닝은 의미상 같은 부류임에도 다른 단어(유사도=0)로 취급
- 스몰데이터에선 성능이 낮을 것이고, 빅데이터에서는 저장공간과 연산시간이 큼
- 직관적이고 간단하므로 실무에서나 단순한 케이스에서 쉽게 사용하는 것 뿐

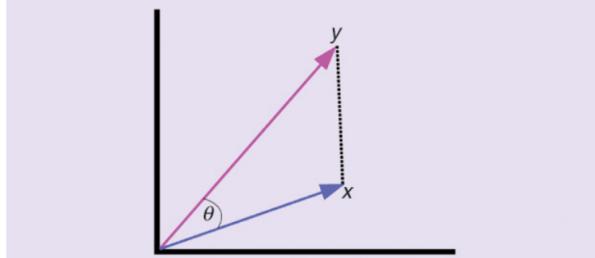
4.2 문서의 유사도

"자연어 문장을 최소의미 단위로 분리하는 형태소 분석 을 넘어 문장의 숨은 의미나 대화의 내용 이해하기 위해서는 동일한 단어 또는 비슷한 단어 가 얼마나 공통적으로 사용되었는지 유사도 추정 필요"

- 유사도의 성능은 어떤 방법으로 수치화(DTM, Word2Vec 등) 또는 계산(유clidean 거리, 코사인 유사도 등) 했는지에 따라 다양하게 존재
- 앞서 Bag of Words 기반 DTM, TF-IDF 도 유사도를 추정하기 위한 하나의 방법이며, 좀 더 정교한 추정을 위해 거리기반 유사도를 많이 사용

1) 유clidean 거리(Euclidean Distance): 두 벡터의 거리를 이용하여 두 벡터가 얼마나 유사한지 측정

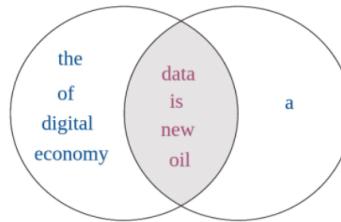
$$\text{Similarity}(x, y) = \|x - y\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$



(<https://www.embs.org/pulse/articles/what-is-the-distance-between-objects-in-a-data-set/>)

2) 자카드 유사도(Jaccard Similarity): 두 벡터의 전체 단어집합에서 공통 단어집합의 비율을 이용하여 두 벡터가 얼마나 유사한지 측정

$$\text{Similarity}(x, y) = J(x, y) = \frac{|x \cap y|}{|x \cup y|} = \frac{|doc_1 \cap doc_2|}{|doc_1 \cup doc_2|}$$



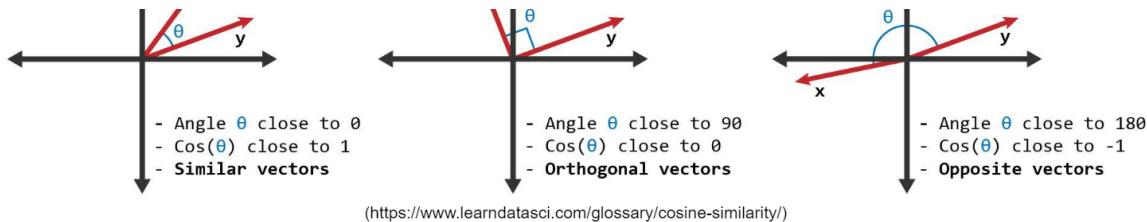
(<https://studymachinelearning.com/jaccard-similarity-text-similarity-metric-in-nlp/>)

3) 코사인 유사도(Cosine Similarity): 두 벡터의 각도를 이용하여 두 벡터가 가리키는 방향이 얼마나 유사한지 측정

$$\text{Similarity}(x, y) = \cos(\theta) = \frac{x \cdot y}{\|x\|\|y\|}$$

- θ 는 두 벡터의 각도
- $x \cdot y = \sum_{i=1}^n x_i y_i = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$ 는 두 벡터의 내적곱
- $\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$ 은 각 벡터의 크기





```
In [20]: # 예시 텍스트 데이터
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords

sentences = "A barber is a person. a barber is good person. #
a barber is huge person. he Knew A Secret! The Secret He Kept is huge secret. #
Huge secret. His barber kept his word. a barber kept his word. #
His barber kept his secret. #
But keeping and keeping such a huge secret to himself was driving the barber crazy. #
the barber went up a huge mountain."
```

단어 토큰화 및 빈도수 추정
stop_words = set(stopwords.words('english'))

```
sentence_pred = []
vocab = dict()
for sentence in sent_tokenize(sentences):
    # 단어 토큰화
    tokenized_sentence = word_tokenize(sentence)
    result = []
    for word in tokenized_sentence:
        # 모든 단어의 소문자화
        word = word.lower()
        # 뚫고 들어 제거
        if word not in stop_words:
            # 단어 길이가 2인하는 제거
            if len(word) > 2:
                result.append(word)
            # 결과 단어 절리 및 빈도수 계산
            if word not in vocab:
                vocab[word] = 0
            vocab[word] = vocab[word] + 1
    sentence_pred.append(result)

print(sentence_pred, '\n\n', vocab)
```

```
[['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], ['keeping', 'huge', 'secret', 'driving', 'barber', 'crazy'], ['barber', 'went', 'huge', 'mountain']]
```

```
{'barber': 8, 'person': 3, 'good': 1, 'huge': 5, 'knew': 1, 'secret': 6, 'kept': 4, 'word': 2, 'keeping': 2, 'driving': 1, 'crazy': 1, 'went': 1, 'mountain': 1}
```

```
In [21]: # 정제된 토큰들의 재운장화
sentence_prep = " ".join(i) for i in sentence_pred
sentence_prep
```

```
Out[21]: ['barber person',
'barber good person',
'barber huge person',
'knew secret',
'secret kept huge secret',
'huge secret',
'barber kept word',
'barber kept word',
'barber kept secret',
'keeping keeping huge secret driving barber crazy',
'barber went huge mountain']
```

```
In [22]: # sklearn 사용 TF-IDF 추출
from sklearn.feature_extraction.text import TfidfVectorizer

tfidifier = TfidfVectorizer(max_features=10)
tfidifier.fit(sentence_prep)
print(tfidifier.vocabulary_)
TF_IDF_sklearn = tfidifier.transform(sentence_prep).toarray()
TF_IDF_sklearn = pd.DataFrame(TF_IDF_sklearn, columns=DTM.columns, index=DTM.index)
TF_IDF_sklearn
```

```
{'barber': 0, 'person': 7, 'good': 3, 'huge': 4, 'secret': 8, 'kept': 6, 'word': 9, 'keeping': 5, 'driving': 2, 'crazy': 1}
```

```
Out[22]:
```

	barber	crazy	driving	good	huge	keeping	kept	person	secret	word
sentence1	0.522986	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.852341	0.000000	0.000000
sentence2	0.345928	0.000000	0.000000	0.74999	0.000000	0.000000	0.000000	0.563780	0.000000	0.000000
sentence3	0.430929	0.000000	0.000000	0.000000	0.566620	0.000000	0.000000	0.702311	0.000000	0.000000
sentence4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
sentence5	0.000000	0.000000	0.000000	0.000000	0.400739	0.000000	0.443892	0.000000	0.801479	0.000000
sentence6	0.000000	0.000000	0.000000	0.000000	0.707107	0.000000	0.000000	0.000000	0.707107	0.000000
sentence7	0.390567	0.000000	0.000000	0.000000	0.000000	0.000000	0.568849	0.000000	0.000000	0.723787
sentence8	0.390567	0.000000	0.000000	0.000000	0.000000	0.000000	0.568849	0.000000	0.000000	0.723787
sentence9	0.454067	0.000000	0.000000	0.000000	0.000000	0.000000	0.661334	0.000000	0.597043	0.000000

```

sentence10 0.174980 0.379366 0.379366 0.00000 0.230078 0.758732 0.000000 0.000000 0.230078 0.000000
sentence11 0.605349 0.000000 0.000000 0.00000 0.795961 0.000000 0.000000 0.000000 0.000000 0.000000

```

In [23]: # 문서의 유사도를 구하는 경우 TF-IDF의 단어행렬 벡터를 비교

```

from numpy import dot
from numpy.linalg import norm

def similarity_cosine(A, B):
    return dot(A, B) / (norm(A) * norm(B))

similarity_cosine(TF_IDF_sklearn.iloc[0,:], TF_IDF_sklearn.iloc[1,:])

```

Out [23]: 0.6614488798177138

In [24]: # 유사도 metrics를 사용하여 TF-IDF의 단어행렬 벡터를 비교

```

from sklearn.metrics.pairwise import euclidean_distances, cosine_similarity
from sklearn.metrics import jaccard_score

```

```

def jaccard_similarity(df_zeroone):
    from itertools import product
    TFIDF_jd = []
    for left, right in product(df_zeroone.index, repeat=2):
        TFIDF_jd.append(jaccard_score(df_zeroone.loc[left], df_zeroone.loc[right]))
    TFIDF_jd = np.array(TFIDF_jd).reshape(df_zeroone.shape[0], df_zeroone.shape[0])
    return pd.DataFrame(TFIDF_jd, columns=df_zeroone.index, index=df_zeroone.index)

TFIDF_ed = pd.DataFrame(euclidean_distances(TF_IDF_sklearn),
                        columns=TF_IDF_sklearn.index, index=TF_IDF_sklearn.index)

TFIDF_jd = jaccard_similarity(TF_IDF_sklearn.astype('bool').astype('int'))

TFIDF_cs = pd.DataFrame(cosine_similarity(TF_IDF_sklearn),
                        columns=TF_IDF_sklearn.index, index=TF_IDF_sklearn.index)

display(TFIDF_ed, TFIDF_jd, TFIDF_cs)

```

	sentence1	sentence2	sentence3	sentence4	sentence5	sentence6	sentence7	sentence8	sentence9	sentence10	sentence11
sentence1	0.000000	0.822862	0.593331	1.414214	1.414214	1.414214	1.261538	1.261538	1.234933	1.347952	1.169112
sentence2	0.822862	0.000000	0.953918	1.414214	1.414214	1.414214	1.315212	1.315212	1.298403	1.370744	1.257452
sentence3	0.593331	0.953918	0.000000	1.414214	1.243329	1.094841	1.289723	1.289723	1.268329	1.260341	0.759118
sentence4	1.414214	1.414214	1.414214	0.000000	0.630113	0.765367	1.414214	1.414214	0.897727	1.240905	1.414214
sentence5	1.414214	1.414214	1.243329	0.630113	0.000000	0.547546	1.222696	1.222696	0.675162	1.202827	1.167071
sentence6	1.414214	1.414214	1.094841	0.765367	0.547546	0.000000	1.414214	1.414214	1.075013	1.161569	0.935062
sentence7	1.261538	1.315212	1.289723	1.414214	1.222696	1.414214	0.000000	0.000000	0.944942	1.365034	1.235776
sentence8	1.261538	1.315212	1.289723	1.414214	1.222696	1.414214	0.000000	0.000000	0.944942	1.365034	1.235776
sentence9	1.234933	1.298403	1.268329	0.897727	0.675162	1.075013	0.944942	0.944942	0.000000	1.251544	1.204269
sentence10	1.347952	1.370744	1.260341	1.240905	1.202827	1.161569	1.365034	1.365034	1.251544	0.000000	1.192429
sentence11	1.169112	1.257452	0.759118	1.414214	1.167071	0.935062	1.235776	1.235776	1.204269	1.192429	0.000000

	sentence1	sentence2	sentence3	sentence4	sentence5	sentence6	sentence7	sentence8	sentence9	sentence10	sentence11
sentence1	1.000000	0.666667	0.666667	0.000000	0.000000	0.000000	0.250	0.250	0.250000	0.142857	0.333333
sentence2	0.666667	1.000000	0.500000	0.000000	0.000000	0.000000	0.200	0.200	0.200000	0.125000	0.250000
sentence3	0.666667	0.500000	1.000000	0.000000	0.200000	0.250000	0.200	0.200	0.200000	0.285714	0.666667
sentence4	0.000000	0.000000	0.000000	1.000000	0.333333	0.500000	0.000	0.000	0.333333	0.166667	0.000000
sentence5	0.000000	0.000000	0.200000	0.333333	1.000000	0.666667	0.200	0.200	0.500000	0.285714	0.250000
sentence6	0.000000	0.000000	0.250000	0.500000	0.666667	1.000000	0.000	0.000	0.250000	0.333333	0.333333
sentence7	0.250000	0.200000	0.200000	0.000000	0.200000	0.000000	1.000	1.000	0.500000	0.125000	0.250000
sentence8	0.250000	0.200000	0.200000	0.000000	0.200000	0.000000	1.000	1.000	0.500000	0.125000	0.250000
sentence9	0.250000	0.200000	0.200000	0.333333	0.500000	0.250000	0.500	0.500	1.000000	0.285714	0.250000
sentence10	0.142857	0.125000	0.285714	0.166667	0.285714	0.333333	0.125	0.125	0.285714	1.000000	0.333333
sentence11	0.333333	0.250000	0.666667	0.000000	0.250000	0.333333	0.250	0.250	0.250000	0.333333	1.000000

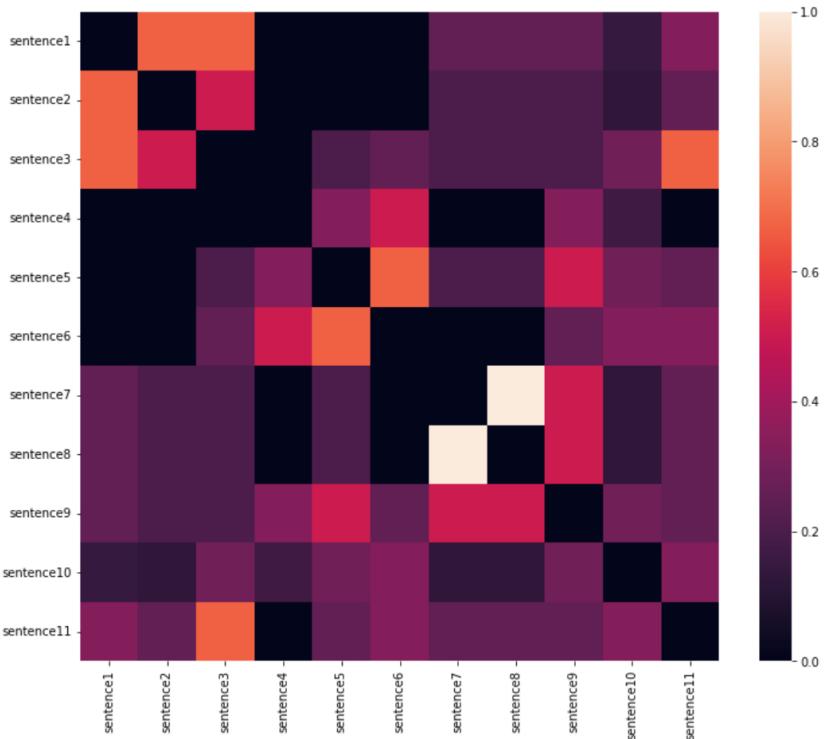
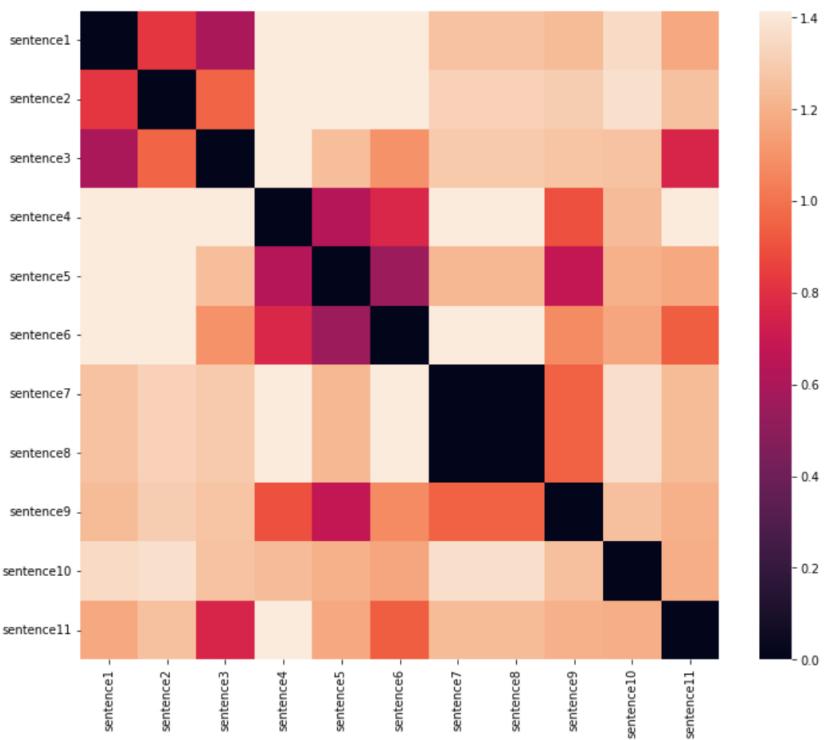
	sentence1	sentence2	sentence3	sentence4	sentence5	sentence6	sentence7	sentence8	sentence9	sentence10	sentence11
sentence1	1.000000	0.661449	0.823979	0.000000	0.000000	0.000000	0.204261	0.204261	0.237471	0.091512	0.316589
sentence2	0.661449	1.000000	0.545020	0.000000	0.000000	0.000000	0.135108	0.135108	0.157075	0.060531	0.209407
sentence3	0.823979	0.545020	1.000000	0.000000	0.227067	0.400661	0.168307	0.168307	0.195671	0.205771	0.711870
sentence4	0.000000	0.000000	0.000000	1.000000	0.801479	0.707107	0.000000	0.000000	0.597043	0.230078	0.000000
sentence5	0.000000	0.000000	0.227067	0.801479	1.000000	0.850097	0.252507	0.252507	0.772078	0.276604	0.318973
sentence6	0.000000	0.000000	0.400661	0.707107	0.850097	1.000000	0.000000	0.000000	0.422173	0.325379	0.562829
sentence7	0.204261	0.135108	0.168307	0.000000	0.252507	0.000000	1.000000	1.000000	0.553543	0.068341	0.236429
sentence8	0.204261	0.135108	0.168307	0.000000	0.252507	0.000000	1.000000	1.000000	0.553543	0.068341	0.236429
sentence9	0.237471	0.157075	0.195671	0.597043	0.772078	0.422173	0.553543	0.553543	1.000000	0.216819	0.274869
sentence10	0.091512	0.060531	0.205771	0.230078	0.276604	0.325379	0.068341	0.068341	0.216819	1.000000	0.289057
sentence11	0.316589	0.209407	0.711870	0.000000	0.318973	0.562829	0.236429	0.236429	0.274869	0.289057	1.000000

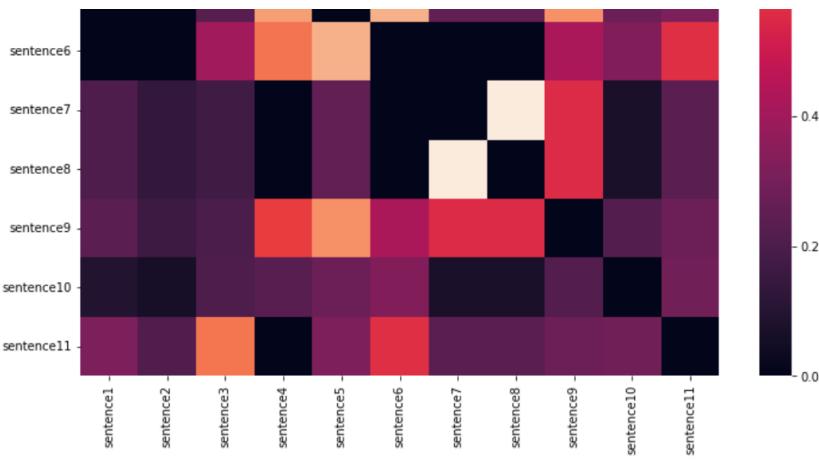
In [25]: # 시각화를 통한 결과 비교
for each in TFIDF_ed, TFIDF_jd, TFIDF_cs:
 # 대각행렬 수치 삭제

```
each.values[inp.diag_indices_from(each)] = 0
```

```
# 결과 시각화
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(12,10))
sns.heatmap(each)
plt.show()
```





4.2.1 유사도기반 BoW 및 TF-IDF 비교

1) 스몰데이터:

```
In [26]: # 임의 문장들의 벡터화
sentences = ['나는 나는 사과를 맛있게 먹었다 먹었다 그리고 그리고 그리고 그리고 그리고 그리고 그리고',
            '나는 나는 바나나를 맛있게 먹었다 먹었다',
            '동생은 사과를 샀다 샀다 그리고 그리고 그리고 그리고 그리고 그리고 그리고 그리고 그리고']

from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer(max_features=10)
tokenizer.fit(sentences)
sentence_encoded = tokenizer.transform(sentences).toarray()
sentence_encoded = sentence_encoded.tolist()
sentence_encoded
```

```
Out[26]: [[10, 2, 0, 1, 2, 0, 1, 0],
           [0, 2, 0, 1, 2, 1, 0, 0],
           [10, 0, 1, 0, 0, 0, 1, 2]]
```

```
In [27]: # 첫번째 문장과 가장 유사한 문장 추출
# 그리고라는 불용어 때문에 첫번째와 세번째 문장의 유사도가 높게 추출
print("문장 A와 문장 B의 유사도: {}".format(round(similarity_cosine(sentence_encoded[0], sentence_encoded[1]), 2)))
print("문장 A와 문장 C의 유사도: {}".format(round(similarity_cosine(sentence_encoded[0], sentence_encoded[2]), 2)))
```

문장 A와 문장 B의 유사도: 0.27
문장 A와 문장 C의 유사도: 0.94

```
In [28]: # 불용어 제거 후의 유사도 비교
# 임의 문장들의 벡터화
sentences = ['나는 나는 사과를 맛있게 먹었다 먹었다',
            '나는 나는 바나나를 맛있게 먹었다 먹었다',
            '동생은 사과를 샀다 샀다']

from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer(max_features=10)
tokenizer.fit(sentences)
sentence_encoded = tokenizer.transform(sentences).toarray()
sentence_encoded = sentence_encoded.tolist()

# 첫번째 문장과 가장 유사한 문장 추출
print("문장 A와 문장 B의 유사도: {}".format(round(similarity_cosine(sentence_encoded[0], sentence_encoded[1]), 2)))
print("문장 A와 문장 C의 유사도: {}".format(round(similarity_cosine(sentence_encoded[0], sentence_encoded[2]), 2)))
```

문장 A와 문장 B의 유사도: 0.9
문장 A와 문장 C의 유사도: 0.13

```
In [29]: # 불용어를 모두 제거할 수만 있다면 유사도 신뢰가능
# 불용어를 제거한다는 것은 분석 목적과 상황에 따라 따라서 거의 불가능
# TF-IDF: 문서를 특징짓는 중요단어는 너무 적지도 많지도 않을 것
sentences = ['나는 나는 사과를 맛있게 먹었다 먹었다 그리고 그리고 그리고 그리고 그리고 그리고',
            '나는 나는 바나나를 맛있게 먹었다 먹었다',
            '동생은 사과를 샀다 샀다 그리고 그리고 그리고 그리고 그리고 그리고 그리고 그리고 그리고 그리고']

from sklearn.feature_extraction.text import TfidfVectorizer

tfidifier = TfidfVectorizer()
tfidifier.fit(sentences)
print(sorted(tfidifier.vocabulary_.items()))
TF_IDF_sklearn = tfidifier.transform(sentences).toarray()
TF_IDF_sklearn = pd.DataFrame(TF_IDF_sklearn)
TF_IDF_sklearn
```

```
[('그리고', 0), ('나는', 1), ('동생은', 2), ('맛있게', 3), ('먹었다', 4), ('바나나를', 5), ('사과를', 6), ('샀다', 7)]
```

```
Out[29]:      0    1    2    3    4    5    6    7
0  0.953463  0.190693  0.000000  0.095346  0.190693  0.000000  0.095346  0.000000
1  0.000000  0.610594  0.000000  0.305297  0.610594  0.401429  0.000000  0.000000
2  0.955007  0.000000  0.125572  0.000000  0.000000  0.095501  0.251144
```

문장 A와 문장 B의 유사도 : 0.26
문장 A와 문장 C의 유사도 : 0.92

2) 빅데이터:

```
In [31]: # 영화 추천을 위한 예제 데이터셋  
import pandas as pd  
  
df = pd.read_csv(r'./Data/Movies/movies_metadata_sample10000.csv', index_col='Unnamed: 0')  
df = df.sample(2000, random_state=123)  
df = df[['title', 'overview']]  
df = df.fillna('').reset_index().iloc[:,1:]  
df
```

	title	overview
0	Vampire Boys	Jasin and his vampire brood's time is running ...
1	Bad Luck Love	A shocking crime forces a man to re-evaluate h...
2	Sicario	A young female FBI agent joins a secret CIA op...
3	Die goldene Gans	
4	Wrong Turn	Chris crashes into a carload of other young pe...
...
1995	Sur	Tina Marie D'Silva lives a middle-class lifest...
1996	Seven Pounds	An IRS agent with a fateful secret embarks on ...
1997	The Arena	Female gladiators fight to the death. Inspired...
1998	Marihuana	A young girl named Burma attends a beach party...
1999	Alice Cooper: Welcome to my Nightmare	A 1975 Alice Cooper concert hosted by Vincent ...

2000 rows × 2 columns

```
In [32]: # 임의 문장들의 벡터화
from sklearn.feature_extraction.text import CountVectorizer

tokenizer = CountVectorizer()
BOW_movie = tokenizer.fit_transform(df['overview']).toarray()
BOW_movie = pd.DataFrame(BOW_movie,
                         columns=[pair[0] for pair in sorted(tokenizer.vocabulary_.items())],
                         index=df.title)

BOW_movie
```

2000 rows × 15367 columns

문장 A와 문장 B의 유사도: 0.42
문장 A와 문장 C의 유사도: 0.32

Out[34]: `[20, 200, 207, 205, 207, 10, 100, 1004b, 101, 105, ...]` (using `PyTorch`)

2000 rows × 15090 columns

문장 A와 문장 B의 유사도: 0.01
문장 A와 문장 C의 유사도: 0.01

4.2.2 유사도 기반 추천시스템

```
In [36]: # 영화 추천을 위한 예제 데이터셋
import pandas as pd

df = pd.read_csv(r'./Data/Movies/movies_metadata_sample10000.csv', index_col='Unnamed: 0')
df = df.sample(2000, random_state=123)
df = df[['title', 'overview']]
df = df.fillna('').reset_index().iloc[:,1:]
df
```

		title	overview
0	Vampire Boys	Jasin and his vampire brood's time is running ...	
1	Bad Luck Love	A shocking crime forces a man to re-evaluate h...	
2	Sicario	A young female FBI agent joins a secret CIA op...	
3	Die goldene Gans		
4	Wrong Turn	Chris crashes into a carload of other young pe...	
...
1995	Sur	Tina Marie D'Silva lives a middle-class lifest...	
1996	Seven Pounds	An IRS agent with a fateful secret embarks on ...	
1997	The Arena	Female gladiators fight to the death. Inspired...	
1998	Marihuana	A young girl named Burma attends a beach party...	
1999	Alice Cooper: Welcome to my Nightmare	A 1975 Alice Cooper concert hosted by Vincent ...	

2000 rows x 2 columns

```
In [37]: # TF-IDF로 줄거리 처리  
# 15090개의 단어로 처리됨  
from sklearn.feature_extraction.text import TfidfVectorizer  
  
tfidifier = TfidfVectorizer(stop_words='english')  
TFIDF_movie = tfidifier.fit_transform(df['overview']).toarray()  
TFIDF_movie = pd.DataFrame(TFIDF_movie,  
                           columns=[pair[0] for pair in sorted(tfidifier.vocabulary_.items())],  
                           index=df.title)  
TFIDF_movie = TFIDF_movie.loc[~TFIDF_movie.index.duplicated(keep='first')]  
TFIDF_movie = TFIDF_movie[TFIDF_movie.columns[TFIDF_movie.sum() != 0]]  
TFIDF_movie
```

Nightmare

1992 rows × 15067 columns

```
In [38]: # sklearn 사용 TF-IDF의 단어행을 벡터를 비교
from sklearn.metrics.pairwise import cosine_similarity

TFIDF_movie = pd.DataFrame(cosine_similarity(TFIDF_movie),
                            columns=list(TFIDF_movie.index), index=list(TFIDF_movie.index))
TFIDF_movie
```

Out [38]:

	Vampire Boys	Bad Luck Love	Sicario	Die goldene Gans	Wrong Turn	The Last Detail	Unforgiven	A Shriek in the Night	The Big Steal	A Cat in Paris	...	The Houses October Built	Shaka Zulu	Extreme Justice	Human Cobras	La Cienaga
Vampire Boys	1.000000	0.008341	0.009962	0.0	0.008968	0.035729	0.004078	0.000000	0.00991	0.0	...	0.005895	0.0	0.000000	0.011454	0.010205
Bad Luck Love	0.008341	1.000000	0.000000	0.0	0.000000	0.013016	0.000000	0.000000	0.000000	0.0	...	0.000000	0.0	0.014222	0.016891	0.012227
Sicario	0.009962	0.000000	1.000000	0.0	0.012364	0.020376	0.000000	0.000000	0.000000	0.0	...	0.000000	0.0	0.052948	0.000000	0.000000
Die goldene Gans	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	...	0.000000	0.0	0.000000	0.000000	0.000000
Wrong Turn	0.008968	0.000000	0.012364	0.0	1.000000	0.050683	0.009261	0.014897	0.000000	0.0	...	0.000000	0.0	0.000000	0.000000	0.000000
...
Sur	0.011841	0.023244	0.000000	0.0	0.000000	0.010737	0.007779	0.000000	0.000000	0.0	...	0.006043	0.0	0.000000	0.009703	0.010392
Seven Pounds	0.000000	0.000000	0.081740	0.0	0.000000	0.000000	0.011760	0.000000	0.000000	0.0	...	0.000000	0.0	0.033286	0.000000	0.000000
The Arena	0.000000	0.000000	0.037520	0.0	0.000000	0.030676	0.000000	0.059853	0.000000	0.0	...	0.000000	0.0	0.000000	0.000000	0.013346
Marijuana	0.011877	0.014491	0.007061	0.0	0.006356	0.010475	0.005535	0.000000	0.000000	0.0	...	0.000000	0.0	0.000000	0.006064	0.007677
Alice Cooper: Welcome to my Nightmare	0.006901	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	...	0.000000	0.0	0.000000	0.000000	0.000000

1992 rows × 1992 columns

```
In [39]: def get_recommendation(title, similarity_metric):
    df_extraction = pd.DataFrame(similarity_metric[title])
    df_extraction.sort_values(by=title, inplace=True, ascending=False)
    df_extraction.columns = ['Top20 Similarity Rank']
    df_extraction = df_extraction.iloc[1:21,:]

    return df_extraction

get_recommendation('The Dark Knight Rises', TFIDF_movie)
```

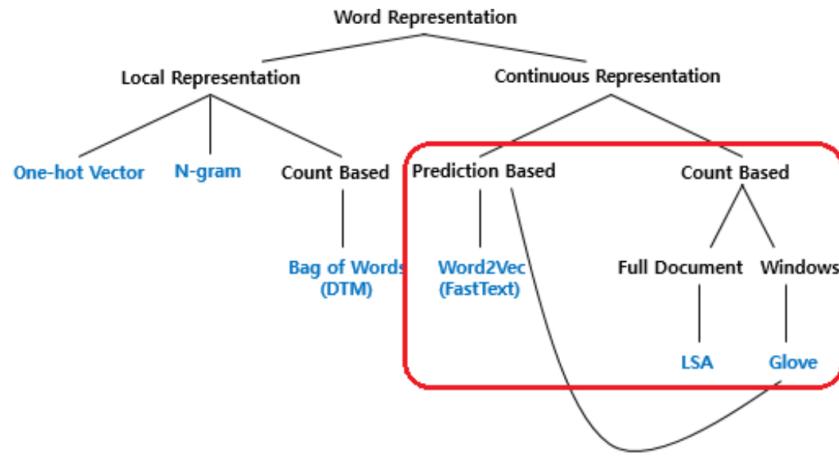
Out [39]:

Top20 Similarity Rank	
Batman Beyond: Return of the Joker	0.221227
Jigsaw	0.120369
Batman Unlimited: Monster Mayhem	0.106685
Stormy Weather	0.082219
Ghosts of Mississippi	0.072593
Dillinger	0.070887
I Thank a Fool	0.069833
The City	0.069439
Justice League vs. Teen Titans	0.066568
The Phenix City Story	0.066276
Zambezia	0.064604
The Kill Team	0.064598
A Knight's Tale	0.064369
11 Blocks	0.063888
Bad Lieutenant	0.062316
Red Dawn	0.061374
Narrow Margin	0.059776
Hieronymus Bosch: Touched by the Devil	0.059364
Captive	0.058098
James White	0.057746

4.3 토픽 모델링

"문서집합에서 단어집합을 찾아 토픽을 구성 함으로써, 우리의 문서가 어떤 토픽 또는 주제인지 군집화(Clustering)하는 텍스트 마이닝 방법론"

- 문서집합의 추상적 주제 발견을 위한 단어통계를 수학적으로 분석하여 추정하는 비지도학습 알고리즘
- 문서 요약 및 분류, 검색결과 최적화 등에 활용



- (1) Singular Value Decomposition(SVD)
- (2) Principal Component Analysis(PCA)
- (3) Latent Semantic Analysis(LSA)
- (4) Probabilistic Latent Semantic Analysis(pLSA)
 - 토픽 모델링의 조상으로 불리는 확률적 모델
 - 새롭게 생성되는 문서를 잘 추정하지 못함
- (5) Latent Dirichlet Allocation(LDA)
 - pLSA의 약점을 개선하기 위해 문서의 주제분포와 주제별 단어분포를 고려
 - 각 토픽들이 나올 확률이 디리클레(Dirichlet) 분포를 따른다는 가정으로 보완

4.3.1 Latent Dirichlet Allocation(LDA)

- 고객은 어떤 내용의 리뷰를 쓸지 확률적 선택하고, 해당 주제의 리뷰에 적합한 단어를 확률적으로 선택하는 과정을 반복하여 문서를 작성한다는 아이디어에 착안
- 확률 기반의 모델링 기법을 통해 방대한 양의 문서 데이터를 분석함으로써 문서 내 어떤 토픽이 어떤 비율로 구성되어 있는지 분석
- 토픽별로 어떤 키워드가 구성되었는지 알려주기 때문에, 키워드 조합을 통해 인사이트를 도출하는 데 효과적
- SNS에서 유사 토픽을 자동으로 분류하거나 항공사 온라인 리뷰를 분석하여 고객 니즈를 도출하는 등 다양한 분야에서 연구 및 활용중

1) 입력 데이터(Input Data): 형태소의 단어의 빈도를 표현하는 DTM(Document Term Matrix)을 입력

- 토픽의 갯수는 분석자가 직접 지정해야 하는 하이퍼파라미터

구분	내용
리뷰1	여긴 삼겹살이랑 냉면 맛집으로 추천
리뷰2	이집 미남 사장님 친절하시고 최고
리뷰3	사장님 친절하시고 삼겹살이랑 된장찌개 맛집이라 추천

2-1) 출력 결과1: 주어진 토픽의 갯수만큼 각 토픽 별 어떤 단어들로 구성되어 있는지 단어의 분포

- 예제는 음식(토픽A)과 사장님(토픽B)과 관련으로 토픽을 구성할 수 있지만, 수천만 건의 문서 데이터에서 토픽 분포를 과학적으로 파악 하려면 반드시 LDA가 필요

토픽 군집화 내용
음식(토픽A) 삼겹살 25%, 냉면 12.5%, 맛집 25%, 추천 25%, 된장찌개 12.5%, 미남 0%, 사장님 0%, 친절 0%, 최고 0%
사장님(토픽B) 삼겹살 0%, 냉면 0%, 맛집 0%, 추천 0%, 된장찌개 0%, 미남 16.5%, 사장님 33%, 친절 33%, 최고 16.5%

2-2) 출력 결과2: 리뷰 별 어떤 토픽들로 구성되어 있는지 토픽의 분포 및 라벨링

구분	내용	토픽 구성	토픽 라벨링
리뷰1	여긴 삼겹살이랑 냉면 맛집으로 추천	토픽A 100%	A
리뷰2	이집 미남 사장님 친절하시고 최고	토픽B 100%	B
리뷰3	사장님 친절하시고 삼겹살이랑 된장찌개 맛집이라 추천	토픽A 60%, 토픽B 40%	A

설치하기

```
!pip install gensim
!pip install pyldavis
```

불러오기

```
from gensim import corpora
from gensim.models import LdaModel, CoherenceModel
import matplotlib.pyplot as plt
from wordcloud import WordCloud
```

```

import pyLDAvis
from pyLDAvis import gensim_models

In [40]: # 설치하기
!pip install gensim
!pip install pyldavis

# 불러오기
import warnings
warnings.filterwarnings('ignore') # 'always'
import pandas as pd
from tqdm import tqdm
from gensim import corpora
from gensim.models import LdaModel, CoherenceModel
import matplotlib.pyplot as plt
from wordcloud import WordCloud
import pyLDAvis
from pyLDAvis import gensim_models

Requirement already satisfied: gensim in c:\Users\kk\anaconda3\lib\site-packages (4.1.2)
Requirement already satisfied: scipy>=0.18.1 in c:\Users\kk\anaconda3\lib\site-packages (from gensim) (1.7.3)
Requirement already satisfied: numpy>=1.17.0 in c:\Users\kk\anaconda3\lib\site-packages (from gensim) (1.21.6)
Requirement already satisfied: smart-open>=1.8.1 in c:\Users\kk\anaconda3\lib\site-packages (from gensim) (5.2.1)

[notice] A new release of pip available: 22.2 -> 22.2.2
[notice] To update, run: python.exe -m pip install --upgrade pip
Requirement already satisfied: pyldavis in c:\Users\kk\anaconda3\lib\site-packages (3.3.1)
Requirement already satisfied: scipy in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (1.7.3)
Requirement already satisfied: scikit-learn in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (1.1.1)
Requirement already satisfied: numexpr in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (2.8.3)
Requirement already satisfied: future in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (0.18.2)
Requirement already satisfied: jinja2 in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (2.11.3)
Requirement already satisfied: gensim in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (4.1.2)
Requirement already satisfied: numpy>=1.20.0 in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (1.21.6)
Requirement already satisfied: pandas>=1.2.0 in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (1.4.3)
Requirement already satisfied: sklearn in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (0.0)
Requirement already satisfied: setuptools in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (61.2.0)
Requirement already satisfied: funcy in c:\Users\kk\anaconda3\lib\site-packages (from pyldavis) (1.17)
Requirement already satisfied: joblib in c:\Users\kk\appdata\roaming\python\python39\site-packages (from pyldavis) (1.0.1)
Requirement already satisfied: python-dateutil>=2.8.1 in c:\Users\kk\anaconda3\lib\site-packages (from pandas>=1.2.0->pyldavis) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\Users\kk\anaconda3\lib\site-packages (from pandas>=1.2.0->pyldavis) (2022.1)
Requirement already satisfied: smart-open>=1.8.1 in c:\Users\kk\anaconda3\lib\site-packages (from gensim>pyldavis) (5.2.1)
Requirement already satisfied: MarkupSafe>=0.23 in c:\Users\kk\appdata\roaming\python\python39\site-packages (from jinja2>pyldavis) (2.0.1)
Requirement already satisfied: packaging in c:\Users\kk\anaconda3\lib\site-packages (from numexpr>pyldavis) (21.3)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\Users\kk\anaconda3\lib\site-packages (from scikit-learn>pyldavis) (2.2.0)
Requirement already satisfied: six>=1.5 in c:\Users\kk\anaconda3\lib\site-packages (from python-dateutil>2.8.1->pandas>=1.2.0->pyldavis) (1.16.0)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in c:\Users\kk\anaconda3\lib\site-packages (from packaging>numexpr>pyldavis) (3.0.4)

[notice] A new release of pip available: 22.2 -> 22.2.2
[notice] To update, run: python.exe -m pip install --upgrade pip

In [41]: # 예제 테이터
sentences=[

    '나는 아침에 라면을 자주 먹는다.',
    '나는 아침에 밥 대신에 라면을 자주 먹는다.',
    '현대인의 삶에서 스마트폰은 필수품이 되었다.',
    '현대인들 중에서 스마트폰을 사용하지 않는 사람은 거의 없다.',
    '점심시간에 스마트폰을 이용해 영어 회화 공부를 하느라 혼자 밥을 먹는다.'
]

stoplist = ('.!?')
texts = [[word for word in document.split() if word not in stoplist]
         for document in sentences]
idx2word = corpora.Dictionary(texts)
corpus = [idx2word.doc2bow(text) for text in texts]

# 토큰 개수에 따라 Coherence score, Perplexity score 계산
def compute_coherence_values(idx2word, corpus, texts, limit, start=2, step=1):
    coherence_values, perplexity_values = [], []
    model_list = []
    for num_topics in range(start, limit, step):
        model = LdaModel(corpus, num_topics=num_topics, id2word=id2word, iterations=50, passes=50)
        model_list.append(model)
        coherence_model = CoherenceModel(model=model, texts=texts, dictionary=id2word, coherence='c_v')
        coherence_values.append(coherence_model.get_coherence())
        perplexity_values.append(LdaModel.log_perplexity(model, chunk=corpus))

    return model_list, coherence_values, perplexity_values

# Coherence score: 값이 클수록 정확한 그룹
# Perplexity score: 값이 작을수록 정확한 그룹
start=2; end=np.shape(corpus)[0]+1; step=1
model_list, coherence_values, perplexity_values = compute_coherence_values(idx2word, corpus, texts,
                                                                           limit=end, start=start, step=step)

plt.plot(range(start, end, step), coherence_values)
plt.xlabel("Num Topics")
plt.ylabel("Coherence score")
plt.legend(("coherence_values"), loc='best')
plt.show()

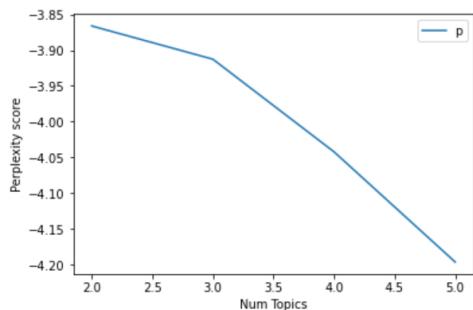
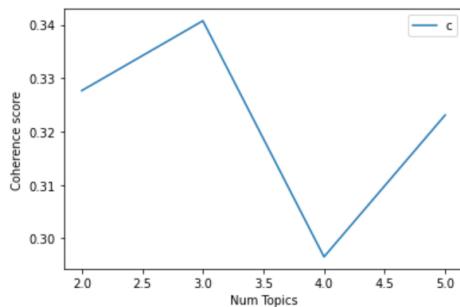
plt.plot(range(start, end, step), perplexity_values)
plt.xlabel("Num Topics")

```

```

plt.ylabel("Perplexity score")
plt.legend(("perplexity_values"), loc='best')
plt.show()

```



```

In [42]: # LDA 모델링
# max(coherence score)로 군집 수 결정
# passes: 딥러닝의 epoch와 같은 개념으로 데이터를 모델에 학습할 횟수
import warnings
warnings.filterwarnings('ignore') # 'always'

num_topics = start + np.argmax(coherence_values)
model_lda = LdaModel(corpus=corpus, id2word=id2word, num_topics=num_topics,
                     passes=5, random_state=123)

# 토픽별 결과 정리
topic_keyword, topic_kwdfreq = dict(), dict()
for topic, keywords in model_lda.show_topics(num_words=20):
    topic_keyword[topic] = [keyword.split('*')[1] for keyword in keywords.split('+')]
    topic_kwdfreq[topic] = [float(keyword.split('*')[1][1:-2]) for keyword in keywords.split('+')]

topic_keyword = pd.DataFrame.from_dict(topic_keyword)
topic_keyword.columns = ['Topic '+str(i+1) for i in range(num_topics)]
display(topic_keyword)

# 빈도수 워드클라우드 시각화
word_cloorder = WordCloud(background_color='white',          # 배경색
                           font_path='C:\Windows\Fonts\malgun.ttf',
                           colormap='autumn',        # 글자컬러맵
                           width=600, height=300,     # 폭과 높이로 figsize랑 맞추어야함
                           random_state=123,          # 랜덤 시각화 고정
                           prefer_horizontal=True,    # 수평글자로 기록
                           max_font_size=100,         # 최대 폰트 크기
                           max_words=20)             # 표현할 최대 단어 갯수

for key, val in topic_kwdfreq.items():
    word_cloud = word_cloorder.generate_from_frequencies(val)
    plt.figure(figsize=(10,8))      # 캔버스 사이즈
    plt.imshow(word_cloud)
    plt.title('Word Frequency by Topic: ' + str(key+1), fontsize=30)    # 제목과 사이즈
    plt.axis('off')               # 그래프 축을 제거
    plt.show()

```

	Topic 1	Topic 2	Topic 3
0	"점심시간에"	"먹는다."	"없다."
1	"밥을"	"나는"	"거의"
2	"스마트폰을"	"아침에"	"않는"
3	"하느라"	"자주"	"중에서"
4	"혼자"	"라면을"	"사람은"
5	"영어"	"씹는"	"현대인들"
6	"회화"	"대신에"	"사용하지"
7	"이용해"	"공부를"	"삶에서"
8	"먹는다."	"스마트폰을"	"필수품이"
9	"공부를"	"이용해"	"현대인의"
10	"라면을"	"회화"	"스마트폰을"
11	"아침에"	"영어"	"스마트폰은"
12	"현대인의"	"혼자"	"되었다."
13	"점심시간에"	"대신에"	"한국인"

13	"피 있나."	"아니다"	"없는나."
14	"자주"	"밥을"	"라면을"
15	"스마트폰은"	"점심시간에"	"자주"
16	"삶에서"	"되었다."	"나는"
17	"필수품이"	"스마트폰은"	"아침에"
18	"나는"	"필수품이"	"공부를"
19	"사용하지"	"현대인의"	"대신에"

Word Frequency by Topic: 1



Word Frequency by Topic: 2



Word Frequency by Topic: 3



- **pyLDAvis:** LDA 모델의 학습결과를 시각적으로 표현 하는 라이브러리로, 차원 축소 방법인 PCA와 키워드추출 방법을 이용하여 토픽 간의 관계와 토픽 키워드를 손쉽게 이해할 수 있도록 지원

(1) Intertopic Distance Map (via multidimensional scaling)

- PCA를 사용해서 다차원의 토픽을 2차원으로 압축
- 원과 원사이의 거리: 토픽 간 서로 얼마나 유사한가
- 원의 크기: 토픽에 해당되는 단어/토큰의 갯수 또는 분포
- 버블 중 하나로 커서를 이동하면 오른쪽의 단어와 막대가 업데이트

(2) Top-30 Most Relevant Terms for Topic

- **막대:** 토픽을 형성하는 주도적인 키워드
- 키워드에 커서를 올리면 해당 키워드와 관련된 토픽 확인
- **λ (Relevant):** 토픽에서 특정 단어의 출현 빈도와 전체 문서에서 그 단어의 출현 빈도를 조정하는 하이퍼파라미터
- $\lambda = 1$: 특정 토픽을 구성하는 키워드보다는 전체 문서 데이터에서 자주 빈출 되는 키워드 일 가능성 높음
- $\lambda = 0$: 전체 문서에서 출현 횟수는 적더라도 해당 토픽을 다른 토픽과 차별성 있게 구분할 수 있는 단어

```
In [43]: # 학습결과 시각화 및 인사이트 확인
from IPython.display import HTML
HTML(filename='visualization_LDA.html')
```

Out [43] :

```
In [44]: # pyLDAvis 시각화
# 하이퍼파라미터는 일반적으로 0.6 정도가 효과적인 것으로 알려져 있으나 정답은 아님
pyLDAvis.enable_notebook()
model_lda_vis = gensim_models.prepare(model_lda, corpus, idx2word)
pyLDAvis.save_html(model_lda_vis, 'visualization_LDA.html')
pyLDAvis.display(model_lda_vis)
```

Out [44]:

```
In [45]: # 문서의 토픽 예측
topic_predict = []
for pred in model_lda.get_document_topics(corpus, minimum_probability=0):
    topic_predict.append(sorted(pred, key=lambda x:x[1], reverse=True)[0])
topic_predict = pd.concat([pd.Series(sentences), pd.DataFrame(topic_predict)], axis=1)
topic_predict.columns = ['Sentences/Documents', 'Predicted Topic', 'Predicted Weight']
topic_predict['Predicted Topic'] = topic_predict['Predicted Topic'] + 1
topic_predict
```

Out [45]:

	Sentences/Documents	Predicted Topic	Predicted Weight
0	나는 아침에 라면을 자주 먹는다.	2	0.887721
1	나는 아침에 밥 대신에 라면을 자주 먹는다.	2	0.915878
2	현대인의 삶에서 스마트폰은 필수품이 되었다.	3	0.888020
3	현대인들 중에서 스마트폰을 사용하지 않는 사람은 거의 없다.	3	0.924594
4	점심시간에 스마트폰을 이용해 영어 회화 공부를 하느라 혼자 밥을 먹는다.	1	0.937768

4.3.2 토픽 기반 추천시스템

```
In [46]: # 영화 추천을 위한 예제 데이터셋
import pandas as pd

df = pd.read_csv(r'..\Data\Movies\movies_metadata_sample10000.csv', index_col='Unnamed: 0')
df = df.sample(2000, random_state=123)
df = df[['title', 'overview']]
df = df.fillna('').reset_index().iloc[:,1:]
df['overview'].sample(10)
```

```
Out [46]: 1116    A British intelligence agent must track down a...
511     An examination of the Soviet slaughter of thou...
560     Criminal on the run, Marion Crane takes refuge...
1432    Alain Leroy is a recovering alcoholic who deci...
1642    Nicolas Philibert goes to America after killin...
976     A group of German boys is ordered to protect a...
959     The images comprise only of material Sergei Lo...
1635    Tells the tale of three buddies in their 20's ...
640     A criminal organization has obtained two nucle...
780     A study of the friendship between a Chinese wo...
Name: overview, dtype: object
```

```
In [47]: # 예제 데이터
sentences = df['overview'].sample(10).tolist()

stoplist = ('.|!?'')
# 불용어 처리
texts = [[word for word in document.split() if word not in stoplist]
         for document in sentences]
idx2word = corpora.Dictionary(texts)
corpus = [idx2word.doc2bow(text) for text in texts]

# 토픽 개수에 따라 Coherence score, Perplexity score 계산
def compute_coherence_values(idx2word, corpus, texts, limit, start=2, step=1):
    coherence_values, perplexity_values = [], []
    model_list = []
    for num_topics in range(start, limit, step):
        model = LdaModel(corpus, num_topics=num_topics, id2word=id2word, iterations=50, passes=50)
        model_list.append(model)
        coherence_model = CoherenceModel(model=model, texts=texts, dictionary=idx2word, coherence='c_v')
        coherence_values.append(coherence_model.get_coherence())
        perplexity_values.append(LdaModel.log_perplexity(model, chunk=corpus))

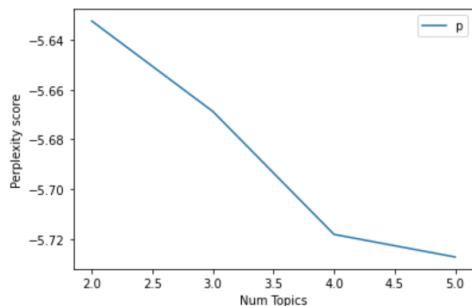
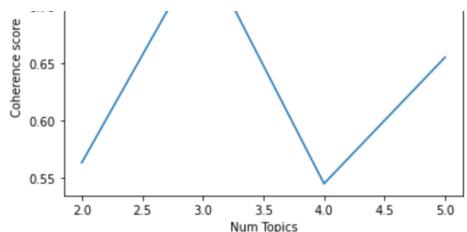
    return model_list, coherence_values, perplexity_values

# Coherence score: 값이 클수록 정확한 그룹
# Perplexity score: 값이 작을수록 정확한 그룹
# start=2; end=np.shape(corpus)[0]+1; step=1
start=2; end=int((np.shape(corpus)[0]/2)+1); step=1
model_list, coherence_values, perplexity_values = compute_coherence_values(idx2word, corpus, texts,
                           limit=end, start=start, step=step)

plt.plot(range(start, end, step), coherence_values)
plt.xlabel("Num Topics")
plt.ylabel("Coherence score")
plt.legend(("coherence_values"), loc='best')
plt.show()

plt.plot(range(start, end, step), perplexity_values)
plt.xlabel("Num Topics")
plt.ylabel("Perplexity score")
plt.legend(("perplexity_values"), loc='best')
plt.show()
```





```
In [48]: # LDA 모델링
# max(coherence score)로 군집 수 결정
# passes: 딥러닝의 epoch와 같은 개념으로 데이터를 모델에 학습할 횟수
import warnings
warnings.filterwarnings('ignore') # 'always'

num_topics = start + np.argmax(coherence_values)
model_lda = LdaModel(corpus=corpus, id2word=id2word, num_topics=num_topics,
                     passes=5, random_state=123)

# 토픽별 결과 정리
topic_keyword, topic_kwdfreq = dict(), dict()
for topic, keywords in model_lda.show_topics(num_words=20):
    topic_keyword[topic] = [keyword.split('*')[1] for keyword in keywords.split('+')]
    topic_kwdfreq[topic] = [float(keyword.split('*')[1][1:-2]) for keyword in keywords.split('+')]

topic_keyword = pd.DataFrame.from_dict(topic_keyword)
topic_keyword.columns = ['Topic '+str(i+1) for i in range(num_topics)]
display(topic_keyword)

# 빈도수 워드클라우드 시각화
word_cloorder = WordCloud(background_color='white',      # 배경색
                           font_path=r'C:\Windows\Fonts\malgun.ttf',
                           colormap='autumn',    # 글자컬러맵
                           width=600, height=300, # 폭과 높이로 figsize랑 맞추어야
                           random_state=123,     # 랜덤 시각화 고정
                           prefer_horizontal=True, # 수평글자로 기록
                           max_font_size=100,    # 최대 폰트 크기
                           max_words=20)        # 표현할 최대 단어 갯수

for key, val in topic_kwdfreq.items():
    word_cloud = word_cloorder.generate_from_frequencies(val)
    plt.figure(figsize=(10,8))    # 캔버스 사이즈
    plt.imshow(word_cloud)
    plt.title('Word Frequency by Topic: ' + str(key+1), fontsize=30)    # 제목과 사이즈
    plt.axis('off')           # 그래프 축을 제거
    plt.show()
```

	Topic 1	Topic 2	Topic 3
0	"the"	"a"	"the"
1	"a"	"the"	"of"
2	"and"	"of"	"to"
3	"with"	"her"	"his"
4	"Inez"	"in"	"Niles"
5	"to"	"to"	"Carol"
6	"in"	"and"	"an"
7	"is"	"she"	"in"
8	"love"	"with"	"Bill"
9	"are"	"on"	"he"
10	"Wonder"	"an"	"a"
11	"Harry"	"at"	"her"
12	"Liane,"	"who"	"has"
13	"of"	"A"	"on"
14	"she"	"has"	"and"
15	"at"	"for"	"up"
16	"wants"	"Fisher"	"takes"
17	"finds"	"as"	"Turkish"
18	"out"	"by"	"building."
19	"A"	"family"	"family"

Word Frequency by Topic: 1

a the Inez in
to Harry Wonder is wants
finds with out are and
Liane, of she at love

Word Frequency by Topic: 2

she with her an
Ahas of as on at
by to for Fisher
in who the and
famil

Word Frequency by Topic: 3

his in to and up
Bill he her of an
building.
Carol the Niles
famil on Turkish takes has

```
In [49]: # pyLDAvis 시작화
# 하이퍼파라미터는 일반적으로 0.6 정도가 효과적인 것으로 알려져 있으나 정답은 아님
pyLDAvis.enable_notebook()
model_lda_vis = gensim_models.prepare(model_lda, corpus, idx2word)
pyLDAvis.save_html(model_lda_vis, 'visualization_LDA.html')
pyLDAvis.display(model_lda_vis)
```

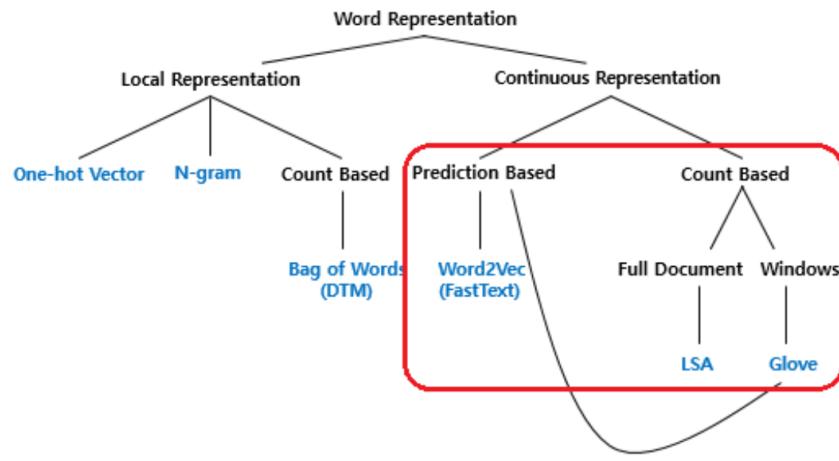
Out [49]:

```
In [50]: # 문서의 토픽 예측
topic_predict = []
for pred in model_lda.get_document_topics(corpus, minimum_probability=0):
    topic_predict.append(sorted(pred, key=lambda x:x[1], reverse=True)[0])
topic_predict = pd.concat([pd.Series(sentences), pd.DataFrame(topic_predict)], axis=1)
topic_predict.columns = ['Sentences/Documents', 'Predicted Topic', 'Predicted Weight']
topic_predict['Predicted Topic'] = topic_predict['Predicted Topic'] + 1
topic_predict
```

Out [50]:

	Sentences/Documents	Predicted Topic	Predicted Weight
0	Tells the story of Fisher Willow, the disliked...	2	0.989123
1		1	0.333333
2	A woman walking home late at night is attacked...	2	0.990466
3	A hypocritical swinging college student cat ra...	2	0.963550
4	Feature length version of the 2013 Turkish sho...	3	0.963375
5	Haru, an aging scriptwriter, has isolated hims...	3	0.981503
6	Ever since she broke up with Nigel, Lena soldi...	2	0.986945
7	Harry and Inez are a dance team at the Wonder ...	1	0.988328
8	A secret admirer's crush on a high school athl...	1	0.949907
9	Niles and Carol Niles are married doctors who ...	3	0.986963

4.4 워드 임베딩(Word Embedding)



- 임베딩(Embedding): 코퍼스에 포함되어 있는 단어들이 각각 하나의 좌표를 가지도록 형성된 벡터 공간

회소 표현(Sparse Representation)	밀집 표현(Dense Representation)
- 원-핫 인코딩처럼 벡터나 행렬의 값이 대부분 0으로 표현	- 0과 1만 가진 값이 아니라 실수 값을 가짐
- 단어의 개수가 늘어나면 벡터의 차원이 커지는 단점	- 벡터의 차원을 단어 집합 크기로 상정하지 않음
- 단어간 유사도 및 의미 표현 불가 (자연어 처리에서 치명적)	- 사용자가 설정한 값으로 모든 단어의 벡터 표현을 맞춤
ex) 강아지 = [00001000...] (1000개의 차원)	ex) 강아지 = [0.2, 1.8, 1.1, -2.1....] (임의 128개 차원)

- 사용자가 설정한 차원 갯수로 조밀하게 표현하는 것을 밀집 표현이라고 함
- 단어를 밀집 표현으로 변환하여 벡터화 하는 것으로 이를 밀집 벡터(Dense Vector), 임베딩 벡터(Embedding Vector)라고 함
- 단어를 랜덤한 값을 가지는 밀집 벡터로 변환 후, 인공신경망이 단어 벡터를 학습
- 유사도를 반영할 수 있도록 벡터화가 되면 연산이 가능 (전화기 + 컴퓨터 = 스마트폰)

원-핫 벡터		임베딩 벡터
차원	고차원 (단어 집합의 크기)	저차원
다른 표현	회소 벡터의 일종	밀집 벡터의 일종
표현 방법	수동	훈련 데이터로부터 학습
값의 타입	0 & 1	실수

- (1) 빈도기반 벡터화: LSA, HAL 등
- (2) 예측기반 벡터화: Word2Vec, FastText, NNLM, PNNLM 등
- (3) 빈도 및 예측기반 벡터화: GloVe 등

4.4.1 Word2Vec 사용법

- 2013년 구글에서 개발한 자연어 처리를 위한 (단일)인공신경망기반 알고리즘
- 유사성 표현을 위해 단어의 의미를 차원 공간에 벡터화 하는 방법인 분산 표현(Distributed Representation)을 사용하고 이렇게 표현된 벡터를 임베딩 벡터(Embedding Vector)라고 함
- 분산표현은 비슷한 단어들은 물려있다(비슷한 문맥에서 등장하는 단어들은 비슷한 의미를 가진다)라는 분포가설을 이용하여 단어들을 학습하고, 벡터에 단어의 의미를 저차원에 분산하여 표현

(1) CBOW(Continuous Bag of Words): 주변단어를 사용하여 중간단어 예측

(2) Skip-Gram: 중간단어를 사용하여 주변단어 예측
▪ 여러 연구들에서 CBOW <<< Skip-Gram이 성능이 좋다고 알려짐

```

In [51]: # 예시 텍스트 데이터
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords

sentences = "A barber is a person. a barber is good person. #
a barber is huge person. he Knew A Secret! The Secret He Kept is huge secret. #
Huge secret. His barber kept his word. a barber kept his word. #
His barber kept his secret. #
But keeping and keeping such a huge secret to himself was driving the barber crazy. #
the barber went up a huge mountain."
  
```

단어 토큰화 및 빈도수 추정

```

stop_words = set(stopwords.words('english'))

sentence_pred = []
vocab = dict()
for sentence in sent_tokenize(sentences):
    # 단어 토큰화
    tokenized_sentence = word_tokenize(sentence)
  
```

```

TOKENIZED_SENTENCE = WORD_TOKENIZER(sentence)
result = []
for word in TOKENIZED_SENTENCE:
    # 모든 단어의 소문자화
    word = word.lower()
    # 불용어 제거
    if word not in stop_words:
        # 단어 길이가 2이하는 제거
        if len(word) > 2:
            result.append(word)
            # 결과 단어 정리 및 빈도수 계산
            if word not in vocab:
                vocab[word] = 0
                vocab[word] = vocab[word] + 1
sentence_pred.append(result)

print(sentence_pred, '\n', vocab)

```

```

[['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], ['keeping', 'keepin', 'huge', 'secret'], ['secret', 'driving', 'barber', 'crazy'], ['barber', 'went', 'huge', 'mountain']]

{'barber': 8, 'person': 3, 'good': 1, 'huge': 5, 'knew': 1, 'secret': 6, 'kept': 4, 'word': 2, 'keeping': 2, 'driving': 1, 'crazy': 1, 'went': 1, 'mountain': 1}

```

In [52]: # 문장의 단어 토큰화 확인
sentence_pred

Out [52]: [['barber', 'person'],
['barber', 'good', 'person'],
['barber', 'huge', 'person'],
['knew', 'secret'],
['secret', 'kept', 'huge', 'secret'],
['huge', 'secret'],
['barber', 'kept', 'word'],
['barber', 'kept', 'word'],
['barber', 'kept', 'secret'],
['keeping', 'keepin', 'huge', 'secret'],
['secret', 'driving', 'barber', 'crazy'],
['barber', 'went', 'huge', 'mountain']]

In [53]: # Word2Vec으로 데이터 훈련시키기
sentences: 문장 토큰화된 데이터
vector_size: 임베딩 할 벡터의 차원
window: 현재값과 예측값 사이의 최대 거리
min_count: 최소 빈도수 제한
worker: 학습을 위한 thread의 수
sg: {0: CBOW, 1: skip-gram}
from gensim.models import Word2Vec

```

model_w2v = Word2Vec(sentences=sentence_pred,
                      vector_size=10, window=5,
                      min_count=2, workers=4, sg=0)

```

In [54]: # 임베딩 결과 크기 확인
model_w2v.wv.vectors.shape

Out [54]: (7, 10)

In [55]: # 임베딩 결과 확인
model_w2v.wv.vectors

Out [55]: array([[-0.00536182, 0.00236439, 0.05103348, 0.09009436, -0.0930305 ,
-0.07116884, 0.06458896, 0.08973141, -0.05015491, -0.03763387],
[0.07380505, -0.01533473, -0.04536615, 0.06554051, -0.0486016 ,
-0.01816018, 0.0287658 , 0.00991874, -0.08285215, -0.09448819],
[0.07311766, 0.05070262, 0.06757693, 0.00762866, 0.06350889,
-0.03405366, -0.00946403, 0.05768573, -0.07521639, -0.03936105],
[-0.07511723, -0.0093006 , 0.09583297, -0.07319304, -0.02333814,
-0.01937778, 0.08077586, -0.05931007, 0.00045162, -0.04753824],
[-0.0960355 , 0.05007293, -0.08759587, -0.04391825, -0.000351 ,
-0.00296183, -0.0766124 , 0.09614742, 0.04982056, 0.09233143],
[-0.08157919, 0.04495797, -0.04137077, 0.00624535, 0.08498619,
-0.04462178, 0.045175 , -0.06786962, -0.03548489, 0.09398508],
[-0.01577654, 0.00321372, -0.0414063 , -0.07682689, -0.01508009,
 0.02469795, -0.0088028, 0.05533662, -0.02742977, 0.02260065],
dtype=float32)

In [56]: # 특정 단어의 벡터 확인
model_w2v.wv['person']

Out [56]: array([-0.0960355 , 0.05007293, -0.08759587, -0.04391825, -0.000351 ,
-0.00296183, -0.0766124 , 0.09614742, 0.04982056, 0.09233143],
dtype=float32)

In [57]: # 입력 단어와 가장 유사한 단어 출력
model_w2v.wv.most_similar('person')

Out [57]: [('word', 0.614398181438446),
('keeping', 0.24953828752040863),
('barber', -0.22418443858623505),
('huge', -0.3207966387271881),
('kept', -0.36627137660980225),
('secret', -0.5381840467453003)]

In [58]: # 입력 단어와 가장 유사한 단어 출력
model_w2v.wv.most_similar('huge')

Out [58]: [('secret', 0.32937225699424744),

```
('barber', 0.30042579770088196),  
('word', -0.08937535434961319),  
('kept', -0.10551013797521591),  
('keeping', -0.151690274477005),  
('person', -0.3207966387271881)]
```

```
In [59]: # 임의 단어와 가장 유사한 단어 출력  
# model_w2v.wv.most_similar('went')
```

```
In [60]: # 임의 두 단어의 유사도 계산  
model_w2v.wv.similarity('person', 'huge')
```

```
Out[60]: -0.32079667
```

```
In [61]: # 학습한 모델 저장 폴더 생성  
import os  
  
folder_location = os.path.join(os.path.dirname(os.getcwd()), '[DataScience]', 'Model', '')  
if not os.path.exists(folder_location):  
    os.makedirs(folder_location)
```

```
In [62]: # 학습한 모델 저장하기  
model_w2v.wv.save_word2vec_format(r'.\Model\model_w2v')
```

```
In [63]: # 학습한 모델 불러오기  
from gensim.models import KeyedVectors  
  
test_w2v = KeyedVectors.load_word2vec_format(r'.\Model\model_w2v')
```

```
In [64]: # 저장한 모델로 임의 두 단어의 유사도 계산  
test_w2v.similarity('person', 'huge')
```

```
Out[64]: -0.32079667
```

```
In [65]: # 각 문장별 변수화로 전체를 빅터화  
model_w2v = Word2Vec(sentences=sentence_pred,  
                     vector_size=5, window=5,  
                     min_count=0, workers=4, sg=0)  
  
for sentence in sentence_pred:  
    print(sentence)  
    vector_word = []  
    for word in sentence:  
        vector_word.append(list(model_w2v.wv[word]))  
    display(vector_word)  
    print()  
  
['barber', 'person']
```

```
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],  
 [0.14623532, 0.10140524, 0.13515386, 0.015257311, 0.12701778]]
```

```
['barber', 'good', 'person']
```

```
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],  
 [-0.03155308, 0.006427431, -0.08281259, -0.15365377, -0.030160189],  
 [0.14623532, 0.10140524, 0.13515386, 0.015257311, 0.12701778]]
```

```
['barber', 'huge', 'person']
```

```
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],  
 [0.14761305, -0.030729944, -0.090678945, 0.13106908, -0.097157314],  
 [0.14623532, 0.10140524, 0.13515386, 0.015257311, 0.12701778]]
```

```
['knew', 'secret']
```

```
[[-0.08924355, 0.09035001, -0.13573924, -0.070969775, 0.18797016],  
 [-0.14235824, 0.12925717, 0.17951462, -0.100218765, -0.075447336]]
```

```
['secret', 'kept', 'huge', 'secret']
```

```
[[-0.14235824, 0.12925717, 0.17951462, -0.100218765, -0.075447336],  
 [-0.03632855, 0.057544768, 0.019833962, -0.16569267, -0.18899609],  
 [0.14761305, -0.030729944, -0.090678945, 0.13106908, -0.097157314],  
 [-0.14235824, 0.12925717, 0.17951462, -0.100218765, -0.075447336]]
```

```
['huge', 'secret']
```

```
[[[0.14761305, -0.030729944, -0.090678945, 0.13106908, -0.097157314],  
 [-0.14235824, 0.12925717, 0.17951462, -0.100218765, -0.075447336]]]
```

```
['barber', 'kept', 'word']
```

```
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],  
 [-0.03632855, 0.057544768, 0.019833962, -0.16569267, -0.18899609],  
 [-0.15024064, -0.018584918, 0.19075581, -0.14637183, -0.046700984]]
```

```
['barber', 'kept', 'word']
```

```
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],  
 [-0.03632855, 0.057544768, 0.019833962, -0.16569267, -0.18899609],  
 [-0.15024064, -0.018584918, 0.19075581, -0.14637183, -0.046700984]]
```

```

['barber', 'kept', 'secret']
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],
 [-0.03632855, 0.057544768, 0.019833962, -0.16569267, -0.1889609],
 [-0.14235824, 0.12925717, 0.17951462, -0.100218765, -0.075447336]
]

['keeping', 'keeping', 'huge', 'secret', 'driving', 'barber', 'crazy']
[[[-0.06808351, -0.018932667, 0.11536362, -0.1503863, -0.0787626],
 [-0.06808351, -0.018932667, 0.11536362, -0.1503863, -0.0787626],
 [0.14761305, -0.030729944, -0.090678945, 0.13106908, -0.097157314],
 [-0.14235824, 0.12925717, 0.17951462, -0.100218765, -0.075447336],
 [-0.16310439, 0.08992348, -0.082773514, 0.016590463, 0.16987504],
 [-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],
 [-0.0059241457, -0.15323071, 0.19229878, 0.09964588, 0.18466328]
]

['barber', 'went', 'huge', 'mountain']
[[-0.0107019255, 0.004702636, 0.10207062, 0.18023966, -0.1860871],
 [-0.19211537, 0.10019175, -0.17516465, -0.08782919, -0.0007403528],
 [0.14761305, -0.030729944, -0.090678945, 0.13106908, -0.097157314],
 [-0.03876416, 0.16161934, -0.1186308, 0.0009527593, -0.09516979]
]

```

```
In [66]: # 유사도 시각화를 위해 차원 변환
from sklearn.manifold import TSNE

X = [list(model_w2v.wv[word]) for word in vocab.keys()]
tsne = TSNE(n_components=2)
X_tsne = tsne.fit_transform(X)
```

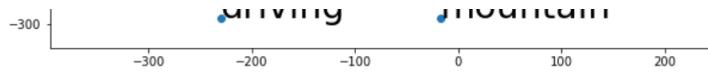
```
In [67]: # 2차원 공간상의 단어 유사도 위치 데이터 정리
X_tsne = pd.DataFrame(X_tsne, index=list(vocab.keys()), columns=['x', 'y'])
```

```
Out [67]:
      x      y
barber -139.343887 -118.390930
person -217.487640  249.969208
good   -339.915314 -108.295914
huge    6.159103   87.077171
knew   -170.210602  57.482544
secret  -19.111126  283.579071
kept    35.747242  -88.871971
word   183.045639 -221.154984
keeping 175.550415  192.632462
driving -230.267593 -294.557800
crazy   -366.230988  94.915077
went    220.404205  -8.058669
mountain -17.453026 -294.747284
```

```
In [68]: # 단어 유사도 시각화
plt.figure(figsize=(10,10))
plt.scatter(X_tsne['x'], X_tsne['y'])

for word, pos in X_tsne.iterrows():
    plt.annotate(word, pos, fontsize=30)
plt.show()
```





4.4.2 Word2Vec 추천시스템 (영어)

```
In [69]: # 영화 추천을 위한 예제 데이터셋
import pandas as pd

df = pd.read_csv(r'..\Data\Movies\movies_metadata_sample10000.csv', index_col='Unnamed: 0')
df = df.sample(2000, random_state=123)
df = df[['title', 'overview']]
df = df.fillna('').reset_index().iloc[:, 1:]
df
```

Out [69]:

	title	overview
0	Vampire Boys	Jasin and his vampire brood's time is running ...
1	Bad Luck Love	A shocking crime forces a man to re-evaluate h...
2	Sicario	A young female FBI agent joins a secret CIA op...
3	Die goldene Gans	
4	Wrong Turn	Chris crashes into a carload of other young pe...
...
1995	Sur	Tina Marie D'Silva lives a middle-class lifest...
1996	Seven Pounds	An IRS agent with a fateful secret embarks on ...
1997	The Arena	Female gladiators fight to the death. Inspired...
1998	Marihuana	A young girl named Burma attends a beach party...
1999	Alice Cooper: Welcome to my Nightmare	A 1975 Alice Cooper concert hosted by Vincent ...

2000 rows × 2 columns

```
In [70]: # TF-IDF로 줄거리 처리
# 15090개의 단어로 처리됨
from sklearn.feature_extraction.text import TfidfVectorizer

tfidfier = TfidfVectorizer(stop_words='english')
TFIDF_movie = tfidfier.fit_transform(df['overview']).toarray()
TFIDF_movie = pd.DataFrame(TFIDF_movie,
                            columns=[pair[0] for pair in sorted(tfidfier.vocabulary_.items())],
                            index=df.title)
TFIDF_movie = TFIDF_movie.loc[~TFIDF_movie.index.duplicated(keep='first')]
TFIDF_movie = TFIDF_movie[TFIDF_movie.columns[TFIDF_movie.sum() != 0]]
TFIDF_movie
```

Out [70]:

	00	000	007	05	07	10	100	100th	101	105	...	évade	in	öllers	über	üçlüyü	şen	şener	настрощик	आव	गत
title																					
Vampire Boys	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Bad Luck Love	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Sicario	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Die goldene Gans	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Wrong Turn	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...
Sur	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Seven Pounds	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
The Arena	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Marihuana	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Alice Cooper: Welcome to my Nightmare	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

1992 rows × 15067 columns

```
In [71]: # 영화별 토큰 정리
TFIDF_prep = TFIDF_movie.astype('bool').astype('int')

TFIDF_token = []
for idx in TFIDF_prep.index:
    TFIDF_token.append([list(TFIDF_prep.T[TFIDF_prep.T[idx] == 1].T.columns)])
```

TFIDF_token

Out [71]:

```
[['angeles',
 'blond',
 'brood',
 'caleb',
 'candidates',
 'college',
 'connection',
 'convince',
 'disrupt',
 'dreaming',
 'enters',
 'eternity',
 'eyes',
 'face',
 'fresh']]
```

```
...  
'gorgeous',  
'idea',  
'instant',  
'jasin',
```

```
In [72]: # Word2Vec으로 데이터 준비시키기  
# sentences: 문장 토큰화된 데이터  
# vector_size: 임베딩 할 벡터의 차원  
# window: 현재값과 예측값 사이의 최대 거리  
# min_count: 최소 빈도수 제한  
# worker: 학습을 위한 thread의 수  
# sg: {0: CBOW, 1: skip-gram}  
from gensim.models import Word2Vec  
  
model_w2v = Word2Vec(sentences=TFIDF_token,  
                     vector_size=100, window=5,  
                     min_count=2, workers=4, sg=0)
```

```
In [73]: # 임베딩 결과 크기 확인  
# 6422개의 단어가 100차원으로 구성  
model_w2v.wv.vectors.shape
```

```
Out[73]: (6422, 100)
```

```
In [74]: # 100차원으로 구성된 단어벡터 확인  
model_w2v.wv.vectors
```

```
Out[74]: array([[-0.06241555,  0.13984482,  0.05805507, ..., -0.15004002,  
                 0.01444292,  0.07384612],  
                [-0.0332361 ,  0.06043577,  0.03149306, ..., -0.05780372,  
                 -0.00768961,  0.02900479],  
                [-0.05476661,  0.13425726,  0.04058682, ..., -0.12932217,  
                 0.02313085,  0.05235415],  
                ...,  
                [ 0.00896209, -0.00271073,  0.0055775 , ..., -0.00133641,  
                 0.00881035, -0.00792186],  
                [-0.00570147, -0.00588458, -0.0012996 , ..., -0.01094489,  
                 0.0013994 ,  0.00115916],  
                [-0.00717888,  0.00922429, -0.00477874, ..., -0.00867292,  
                 0.00194297,  0.00098737], dtype=float32)
```

```
In [75]: # 입력 단어와 가장 유사한 단어 출력  
model_w2v.wv.most_similar('hero')
```

```
Out[75]: [('death', 0.9514639377593994),  
          ('life', 0.9491758942604065),  
          ('family', 0.9488354325294495),  
          ('film', 0.9482184648513794),  
          ('high', 0.9468833208084106),  
          ('meets', 0.9468768835067749),  
          ('home', 0.9468110203742981),  
          ('house', 0.9467649459838867),  
          ('goes', 0.9460838437080383),  
          ('daughter', 0.9460544586181641)]
```

```
In [76]: # 입력 단어와 가장 유사한 단어 출력  
model_w2v.wv.most_similar('male')
```

```
Out[76]: [('playing', 0.6908953785896301),  
          ('married', 0.6851315498352051),  
          ('crime', 0.683753490447998),  
          ('ex', 0.6813966035842896),  
          ('forced', 0.6783660650253296),  
          ('begin', 0.6756349802017212),  
          ('detective', 0.6736080646514893),  
          ('revenge', 0.6734219789505005),  
          ('plan', 0.6733288764953613),  
          ('german', 0.6709856986999512)]
```

```
In [77]: # 입력 단어와 가장 유사한 단어 출력  
model_w2v.wv.most_similar('female')
```

```
Out[77]: [('help', 0.9503719806671143),  
          ('mother', 0.9489347338676453),  
          ('man', 0.9485166072845459),  
          ('day', 0.9481666684150696),  
          ('lives', 0.9477668404579163),  
          ('night', 0.9476389288902283),  
          ('movie', 0.9475988149642944),  
          ('new', 0.9474319815635681),  
          ('old', 0.9473823308944702),  
          ('brother', 0.9473381638526917)]
```

```
In [78]: # 입력 두 단어의 유사도 계산  
model_w2v.wv.similarity('hero', 'male')
```

```
Out[78]: 0.6072017
```

```
In [79]: # 입력 두 단어의 유사도 계산  
model_w2v.wv.similarity('hero', 'female')
```

```
Out[79]: 0.90037847
```

```
In [80]: # 각 문장별 변수화로 전체글 벡터화  
model_w2v = Word2Vec(sentences=TFIDF_token,  
                     vector_size=100, window=5,
```


- [Embedding Projector](#): 구글이 지원하는 데이터 시각화 도구

4.4.3 Word2Vec 추천시스템 (한국어)

```
In [85]: # 추천을 위한 리뷰댓글 예제 데이터셋
import pandas as pd

df = pd.read_table(r'Data\NaverMovieReview\ratings.txt')
df = df.dropna(how='any')
df['document'] = df['document'].str.replace("[^ㄱ-ㅎㅏ-ㅣ가-힣]", "")
df = df.sample(10000, random_state=123)
df
```

```
Out [85]:
      id                               document  label
53310  4448871  너무 재미있어서 다시 보고 싶다    1
126914 9985601 정말 말그대로 킬링타임용 지하철버스타면서 핸드폰으로 보는 용도로 딱 이태란언니 단아...
66133   2616109           진짜 이렇게 집중해서 볼 수 있는 드라마 보기힘들다 ㅎ    1
31387   5918692           이해할 수 없는평점 난 너무 좋았는데 승혜교의 연기가 돌보였다    1
165817   8146857           전기세가 아깝다    0
...     ...
70111  2090804           김현정이 귀여워서 점준다    1
165365 1369471           배우들만 멋지군    0
19996   9988389           솔까이게 절대 영화는 아닌것 같은데    1
180890  4028297           이게재밋니 ㅋㅋㅋ    0
2874   812739           시간이넘는시간동안 눈을 뗄수 없었다    1
```

10000 rows × 3 columns

```
In [86]: # KoNLPy 설치
import os
print('JAVA_HOME' in os.environ)
import sys
print(sys.version)
os.environ['JAVA_HOME'] = r'C:\Program Files\Java\jdk-18.0.2.1\bin\server'
print('JAVA_HOME' in os.environ)
!python -m site
!pip install konlpy
```

```
False
3.9.7 (default, Sep 16 2021, 16:59:28) [MSC v.1916 64 bit (AMD64)]
True
sys.path = [
    'C:\DataScience\Lecture\【DataScience】',
    'C:\Users\Kk\anaconda3\python39.zip',
    'C:\Users\Kk\anaconda3\DLLs',
    'C:\Users\Kk\anaconda3\lib',
    'C:\Users\Kk\anaconda3',
    'C:\Users\Kk\AppData\Roaming\Python\Python39\site-packages',
    'C:\Users\Kk\anaconda3\lib\site-packages',
    'C:\Users\Kk\anaconda3\lib\site-packages\win32',
    'C:\Users\Kk\anaconda3\lib\site-packages\win32\lib',
    'C:\Users\Kk\anaconda3\lib\site-packages\Pythonwin',
]
USER_BASE: 'C:\Users\Kk\AppData\Roaming\Python' (exists)
USER_SITE: 'C:\Users\Kk\AppData\Roaming\Python\Python39\site-packages' (exists)
ENABLE_USER_SITE: True
Requirement already satisfied: konlpy in c:\users\kk\anaconda3\lib\site-packages (0.6.0)
Requirement already satisfied: konlpy in c:\users\kk\anaconda3\lib\site-packages (0.6.0)
```

```
In [87]: # 불러오기
from konlpy.tag import *

hannanum = Hannanum()
kkma = Kkma()
komoran = Komoran()
okt = Okt()
# mecab = Mecab()
```

```
In [88]: # 형태소 분석기 OKT를 사용한 토큰화 작업 (다소 시간 소요)
from tadm import tadm
stopwords = ['의', '가', '이', '은', '들', '는', '좀', '잘', '걍', '과', '도', '를', '으로', '자', '에', '와', '한', '하다']

tokenized = []
for sentence in tadm(df['document']):
    tokenized_sentence = okt.morphs(sentence) # 토큰화
    stopwords_removed_sentence = [word for word in tokenized_sentence if not word in stopwords] # 불용어 제거
    tokenized.append(stopwords_removed_sentence)

100%
```

```
In [89]: # 리뷰 길이 분포 확인
import matplotlib.pyplot as plt

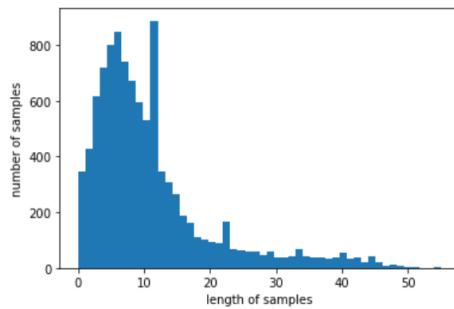
print('리뷰의 최대 길이 : ', max(len(review) for review in tokenized))
```

```

print('리뷰의 평균 길이 : ',sum(map(len, tokenized))/len(tokenized))
plt.hist([len(review) for review in tokenized], bins=50)
plt.xlabel('length of samples')
plt.ylabel('number of samples')
plt.show()

```

리뷰의 최대 길이 : 55
리뷰의 평균 길이 : 10.9604



```

In [90]: # Word2Vec으로 데이터 훈련시키기
# sentences: 문장 토큰화된 데이터
# vector_size: 임베딩 할 벡터의 차원
# window: 현재값과 예측값 사이의 최대 거리
# min_count: 최소 빈도수 제한
# worker: 학습을 위한 thread의 수
# sg: {0: CBOW, 1: skip-gram}
from gensim.models import Word2Vec

model_w2v = Word2Vec(sentences=tokenized,
                     vector_size=100, window=5,
                     min_count=2, workers=4, sg=0)

```

```

In [91]: # 임베딩 결과 크기 확인
# 5130개의 단어가 100차원으로 구성
model_w2v.wv.vectors.shape

```

Out[91]: (8018, 100)

```

In [92]: # 입력 단어와 가장 유사한 단어 출력
model_w2v.wv.most_similar('최민수')

```

Out[92]: [('새로운', 0.9905195832252502), ('대', 0.990274965763092), ('어린이', 0.9902517199516296), ('캐스팅', 0.9901911020278931), ('상영', 0.9901887774467468), ('조반', 0.9901807904243469), ('이따위', 0.9901635646820068), ('비교', 0.9901599884033203), ('개그', 0.9901322722434998), ('그래서', 0.9901157021522522)]

```

In [93]: # 입력 단어와 가장 유사한 단어 출력
model_w2v.wv.most_similar('김수현')

```

Out[93]: [('대의', 0.9427024722099304), ('장국영', 0.942570686340332), ('느끼친다', 0.9408314228057861), ('건물', 0.9402849078178406), ('나루토', 0.939926028251648), ('무한도전', 0.9397677183151245), ('실지', 0.939386785030365), ('흘', 0.9392635822296143), ('제이슨', 0.9392557144165039), ('다수', 0.9392502903938293)]

```

In [94]: # 임의 두 단어의 유사도 계산
model_w2v.wv.similarity('최민수', '김수현')

```

Out[94]: 0.93301946

```

In [95]: # 각 문장별 변수화로 전체를 벡터화
model_w2v = Word2Vec(sentences=tokenized,
                     vector_size=100, window=5,
                     min_count=0, workers=4, sg=0)

for sentence in tokenized:
    vector_word = []
    for word in sentence:
        vector_word.append(list(model_w2v.wv[word]))

```

```

In [96]: # 유사도 시각화를 위해 차원 변환
from sklearn.manifold import TSNE

X = [list(model_w2v.wv[word]) for word in sum(tokenized, [])]
tsne = TSNE(n_components=2)
X_tsne = tsne.fit_transform(X)

```

```

In [97]: # 2차원 공간상의 단어 유사도 위치 데이터 정리
X_tsne = pd.DataFrame(X_tsne, index=sum(tokenized, []), columns=['x', 'y'])
Y_tsne

```

	x	y
너무	-22.539261	18.119392
재미있어서	26.440121	18.014807
다시	40.002731	39.425373
보고	42.446789	0.522534
싶다	-16.465714	35.226246
...
동안	-22.136171	-27.217548
눈	-22.686188	22.904119
을	-5.398875	18.212793
뗄수	-3.275720	-7.567635
없었다	-36.780785	-18.746336

109604 rows × 2 columns

```
[98]: # 단어 유사도 시각화
# 한글을 사용하려면 본인 PC에 설치된 폰트의 종류를 알아야 함
# 본인 PC에 설치되지 않은 폰트는 사용할 수 없음
# 설치된 font 하나 선택 후 mpl 내부 rc 함수에 입력값으로 사용하여 그림 전체에 적용 가능
import matplotlib as mpl
mpl.rcParams['axes'].unicode_minus=False)
import matplotlib.font_manager

fontlist = [font.name for font in matplotlib.font_manager.FontManager.ttflist]

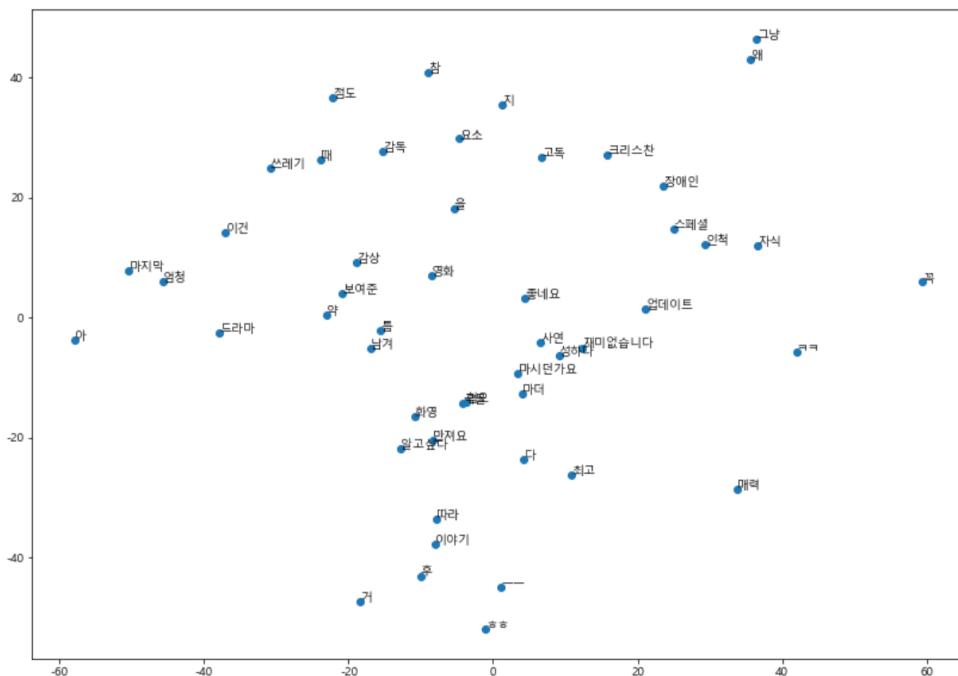
# window
if 'Malgun Gothic' in fontlist:
    mpl.rcParams['font'].family='Malgun Gothic')

# mac
if 'AppleGothic' in fontlist:
    mpl.rcParams['font'].family='AppleGothic')

# 시각화
X_tsne_sub = X_tsne.sample(50, random_state=123)

plt.figure(figsize=(14,10))
plt.scatter(X_tsne_sub['x'], X_tsne_sub['y'])

for word, pos in X_tsne_sub.iterrows():
    plt.annotate(word, pos, fontsize=10)
plt.show()
```



```
In [99]: # Embedding Projector 시각화
from gensim.models import KeyedVectors

model_w2v.wv.save_word2vec_format('visualization_w2v')
!python -m gensim.scripts.word2vec2tensor --input visualization_w2v --output visualization_w2v

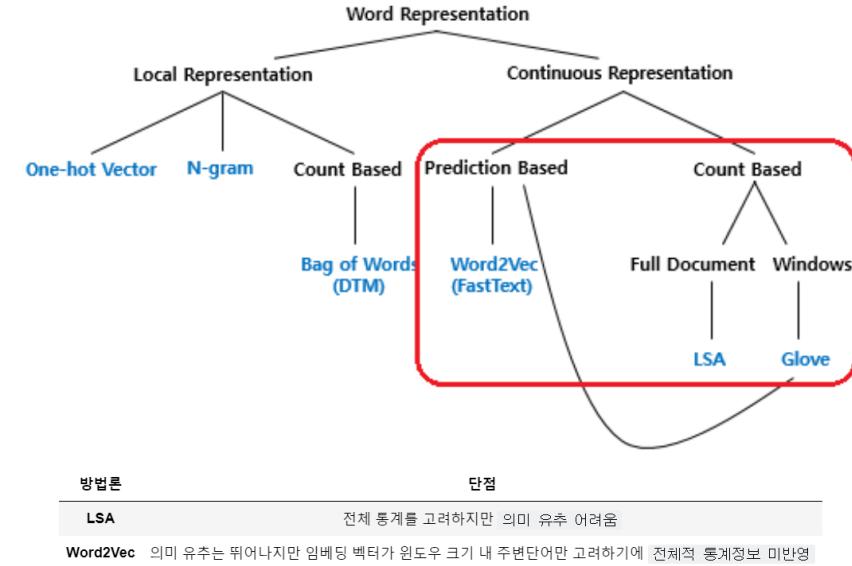
2022-09-15 15:58:15,008 - word2vec2tensor - INFO - running C:\Users\KKK\anaconda3\lib\site-packages\gensim\scripts\word2vec2tensor.py
--input visualization_w2v --output visualization_w2v
2022-09-15 15:58:15,009 - keyedvectors - INFO - loading projection weights from visualization_w2v
2022-09-15 15:58:16,169 - utils - INFO - KeyedVectors lifecycle event {'msg': 'loaded (20590, 100) matrix of type float32 from visualization_w2v', 'binary': False, 'encoding': 'utf8', 'datetime': '2022-09-15T15:58:16.153350', 'gensim': '4.1.2', 'python': '3.9.7 (def
ault, Sep 16 2021, 16:16:04) [GCC 10.2.0 (Aldor-LLVM)]', 'version': '4.1.2'}  
In [100]: !npx tensorboard dev --logdir=visualization_w2v
```

```
2022-09-15 15:58:17,537 - word2vec2tensor - INFO - 2D tensor file saved to visualization_w2v_tensor.tsv
2022-09-15 15:58:17,537 - word2vec2tensor - INFO - Tensor metadata file saved to visualization_w2v_metadata.tsv
2022-09-15 15:58:17,538 - word2vec2tensor - INFO - finished running word2vec2tensor.py
```

- [Embedding Projector](#): 구글이 지원하는 데이터 시각화 도구

4.4.4 GloVe 사용법

"예측과 빈도를 모두 사용하는 방법론으로 2014년 미국 스텐포드대학에서 개발된 임베딩 방법론"



방법론

LSA

단점

전체 통계를 고려하지만 의미 유추 어려움

Word2Vec 의미 유추는 뛰어나지만 임베딩 벡터가 원도우 크기 내 주변단어만 고려하기에 전체적 통계정보 미반영

- 빈도기반 LSA(Latent Semantic Analysis)와 예측기반 Word2Vec의 단점을 보완하기 위한 목적
- Word2Vec과 GloVe 중 어떤 것이 뛰어나다 말하긴 어렵고, 둘다 사용하여 성능이 높은 것을 선택

설치하기

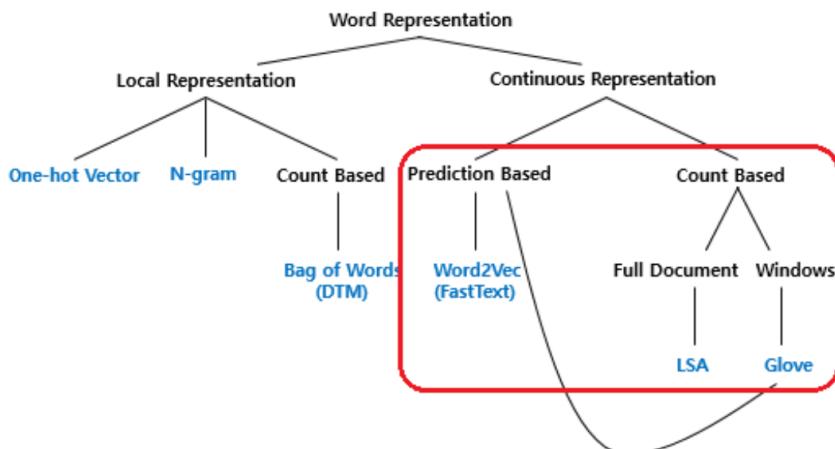
```
!pip install glove-python-binary
```

불러오기

```
import glove
from glove import Glove, Corpus
```

4.4.5 FastText 사용법

"Word2Vec의 확장으로 2017년 페이스북에서 개발된 임베딩 방법론"



- Word2Vec은 단어가 조합수 없는 단위로 가정, FastText는 하나의 단어 안에도 여러 단어들 존재 가정
- 단어의 내부 단어(Subword)를 N-gram으로 고려하여 Word2Vec으로 학습

▪ apple = <ap + app + ppl + ppl + le> + <app + appl + pple + ple> + <appl + pple> + , . . . , + <apple>

- 내부 단어를 통해 모르는 단어(Out Of Vocabulary, OOV)에 대해서도 다른 단어와 유사도 계산 가능 (Word2Vec & GloVe 불가)

▪ `dimnplace = dimin + place`

- Word2Vec이 빈도수가 적은 단어(Rare Word)에서 임베딩 정확도가 낮은데, FastText는 N-gram이 다른 단어의 N-gram과 겹치면 비교적 높은 임베딩 벡터값

- 데이터에 오타나 맞춤법이 틀린 단어(Rare Word)가 존재할 수밖에 없는데, Word2Vec은 임베딩 어려움
- `apple, apple`

불러오기

```
from gensim.models import FastText
```

```
In [100]: # 주제를 위한 리뷰들을 예제 데이터셋
import pandas as pd

df = pd.read_table('..\Data\NaverMovieReview\ratings.txt')
df = df.dropna(how='any')
df['document'] = df['document'].str.replace("[^ㄱ-ㅎㅏ-ㅣ가-힣 ]", "")
df = df.sample(10000, random_state=123)

# KonLPy 설치
import os
print('JAVA_HOME' in os.environ)
import sys
print(sys.version)
os.environ['JAVA_HOME'] = r'C:\Program Files\Java\jdk-18.0.2.1\bin\server'
print('JAVA_HOME' in os.environ)
!python -m site
!pip install konlpy

# 형태소 분석기 OKT를 사용한 토큰화 작업 (다소 시간 소요)
from tadm import tadm
stopwords = ['의', '가', '이', '은', '들', '는', '좀', '잘', '강', '과', '도', '를', '으로', '자', '에', '와', '한', '하다']

tokenized = []
for sentence in tadm(df['document']):
    tokenized_sentence = okt.morphs(sentence) # 토큰화
    stopwords_removed_sentence = [word for word in tokenized_sentence if not word in stopwords] # 불용어 제거
    tokenized.append(stopwords_removed_sentence)
```

```
True
3.9.7 (default, Sep 16 2021, 16:59:28) [MSC v.1916 64 bit (AMD64)]
True
True
sys.path =
['C:\DataScience\#Lecture\#DataScience',
'C:\Users\K\anaconda3\python39.zip',
'C:\Users\K\anaconda3\DLLs',
'C:\Users\K\anaconda3',
'C:\Users\K\anaconda3\Python\Python39\site-packages',
'C:\Users\K\anaconda3\lib\site-packages',
'C:\Users\K\anaconda3\lib\site-packages\win32',
'C:\Users\K\anaconda3\lib\site-packages\win32\lib',
'C:\Users\K\anaconda3\lib\site-packages\Pythonwin',
]

USER_BASE: 'C:\Users\K\AppData\Roaming\Python' (exists)
USER_SITE: 'C:\Users\K\AppData\Roaming\Python\Python39\site-packages' (exists)
ENABLE_USER_SITE: True
Requirement already satisfied: konlpy in c:\users\k\anaconda3\lib\site-packages (0.6.0)
Requirement already satisfied: JPyte>=0.7.0 in c:\users\k\anaconda3\lib\site-packages (from konlpy) (1.4.0)
Requirement already satisfied: lmpl>=4.1.0 in c:\users\k\anaconda3\lib\site-packages (from konlpy) (4.9.1)
Requirement already satisfied: numpy>=1.6 in c:\users\k\anaconda3\lib\site-packages (from konlpy) (1.21.6)

[notice] A new release of pip available: 22.2 -> 22.2.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

100% |██████████| 10000/10000 [00:16<00:00, 597.03it/s]

```
In [101]: # Word2Vec으로 데이터 훈련시키기
from gensim.models import Word2Vec
```

```
model_w2v = Word2Vec(sentences=tokenized,
                      vector_size=100, window=5,
                      min_count=2, workers=4, sg=0)
```

```
# FastText로 데이터 훈련시키기
from gensim.models import FastText
```

```
model_ft = FastText(sentences=tokenized,
                     vector_size=100, window=5,
                     min_count=2, workers=4, sg=0)
```

```
In [102]: # Word2Vec vs FastText 유사한 단어 출력
display('Word2Vec', model_w2v.wv.most_similar('최민수'), 'FastText', model_ft.wv.most_similar('최민수'))
```

'Word2Vec'

```
[('새로운', 0.9936309456825256),
 ('대', 0.9934816956520081),
 ('비교', 0.9934604167938232),
 ('초반', 0.9934467077255249),
 ('좋은', 0.9934457540512085),
 ('함', 0.9934167861938477),
 ('정', 0.9934055209159851),
 ('캐스팅', 0.9933892488479614),
 ('재밌게', 0.9933826923370361),
 ('보면', 0.993378758430481)]
```

```
'FastText'  
[('로맨틱 코미디', 0.9990555644035339),  
 ('힘들었던', 0.99905027961731),  
 ('이건', 0.9990239143371582),  
 ('만들자', 0.9990202784538269),  
 ('봤는데요', 0.9990193247795105),  
 ('봤는데도', 0.9990178346633911),  
 ('흥', 0.999016061286926),  
 ('재밌긴', 0.9990020394325256),  
 ('네이버', 0.9990007877349854),  
 ('않다', 0.9990004301071167)]
```

```
In [103]: # # Word2Vec 유사한 단어 출력  
# print('Word2Vec')  
# model_w2v.wv.most_similar('최만수')
```

```
In [104]: # FastText 유사한 단어 출력  
# Word2Vec이 학습하지 못한 단어도 FastText는 유사단어를 계산하여 출력  
print('FastText')  
model_ft.wv.most_similar('최만수')
```

```
FastText  
Out [104]: [('할것', 0.23927627503871918),  
 ('고름', 0.23108279705047607),  
 ('느꼈다', 0.23085926473140717),  
 ('의존', 0.21700982749462128),  
 ('여장', 0.2163342386484146),  
 ('장비', 0.21452535688877106),  
 ('모델', 0.21277600526809692),  
 ('회때', 0.21227699518203735),  
 ('핵', 0.2122366577386856),  
 ('양쪽', 0.2091122269630432)]
```

```
In [ ]:
```