

# **Text Mining**

**Bok, Jong Soon**

**[javaexpert@nate.com](mailto:javaexpert@nate.com)**

**<https://github.com/swacademy>**

# Text Mining



# Text Mining개요

- Text Data 접근(Text-as-Data Approach)
  - Digital化된 Text를 수치형 Data로 간주하여 Algorithm을 적용해서 분석/요약하는 통계기법
- 위계적 질서를 갖는 Text를 Row와 Column의 Matrix 형태를 갖는 Data로 개념화.
  - 문자(Letter) → 형태소(Morpheme) → 단어(Word) → 문장(Sentence) → 단락(Paragraph) → 문서(Document) → 말뭉치(Corpus)
- Text Data를 이해하고 모형화하는 작업
- Token을 근거로 문서에 나타난 주제(Topic)이나 감정(Sentiment)을 추정.

# Text Mining개요(Cont.)

## ■ Data Mining

- 대규모 Big Data(정형화된 Data)에서 가치 있는/의미 있는 정보를 추출하는 기술
- 사람이 예측할 수 없는 의미 있는 경향과 규칙까지 발견하기 위해서 대량의 Big Data로부터 자동화 또는 반자동화 도구를 활용해 탐색하고 분석하는 과정
- 고급 통계 분석과 Modeling 기법을 적용하여 Data 안의 Pattern과 관계를 찾아내는 과정.

# Text Mining개요(Cont.)

## ■ Text Mining

- 대규모의 Text 문서에서 의미 있는 추출하는 기술
- 비정형 Data(글자/텍스트)를 정형화 및 특징을 추출하는 과정이 필요.
- Text 덩어리 안에서 단어들을 분해해 단어의 출현빈도나 단어들 간의 관계성을 파악하여 의미 있는 정보를 추출해내는 기술
- 정보 검색, Data Mining, Machine Learning, 통계학, Computer 언어학(Computational linguistics) 및 Computer 공학 등이 결합된 분야
- 특히 분석 대상이 형태가 일정하지 않고 다루기 힘든 비정형 데이터이므로 인간의 언어를 Computer가 인식해 처리하는 자연어 처리(NLP) 방법과 관련이 깊음.

# Text Mining개요(Cont.)

- 전통적 내용분석 방식이 유지되기 어려운 이유
  - Social Media, Online 공간에서 기존과는 다른 새로운 Text가 넘쳐나고 있다.
  - 내용분석을 실시할 coder들을 고용하고 교육시키기 위한 비용과 시간이 더욱 많이 필요.
  - Coder들은 인간이기 때문에 검사-재검사 신뢰도(Test-Retest Reliability) 낮다.
  - 방대한 Text를 분석하는 데 엄청난 시간과 비용이 소요.

# Text Mining 개요(Cont.)

## ■ “It is good” vs “It is bad”

	bad	good	is	it	Sentiment
“It is good”	0	1	1	1	?
“It is bad”	1	0	1	1	?

## ■ Text-as-Data

- Data Matrix
- Numerical Array : 분석 단위 X 단어
- Document-Term Matrix(DTM) : 문서 X 단어 행렬

# Text Mining개요(Cont.)

- Text Data / DTM(문서 X 단어 행렬, Document-Term Matrix)를 분석하는 방법
  - Dictionary-based Approach
    - 사전(事前, A Priori)에 규정된 단어의 의미를 기반으로 Text의 의미 추정.
    - Computer Algorithm을 이용한 Text Mining
    - 감정분석(Sentiment Analysis)
  - Machine Learning을 이용한 기법
    - 단어의 의미는 기계학습 사후(事後, A Post Hoc)에 추정.

# Text Mining개요(Cont.)

## ■ 지도 기계학습(Supervised Machine Learning)

- 기계학습을 위해 사용되는 Text Data 중 일부에서 Text의 의미가 알려져 있는 경우
- 훈련 Data는 예측변수와 결과변수로 구성
- 예측변수와 단어들의 속성(예:빈도 ; Frequency 등)과 이 예측변수를 통해 예측되는 결과변수인 Text Data의 분석 단위에 나타난 의미(예:Topic이나 Sentiment 등)으로 구성.
- 예측변수와 결과변수의 관계를 최적으로 설명할 수 있는 함수 추출.
- 문서에 표출된 감정이 어떤지에 대한 인간의 판단과 문서에 등장하는 단어들의 관계를 지도학습을 통해 추정한 후, 새로운 문서에 지도학습으로 추정된 문서분류모형을 적용
- Logistic Regression, Naïve Bayes Classification, SVM, Boosting, Deep Learning

# Text Mining개요(Cont.)

## ■ 비지도 기계학습(Unsupervised Machine Learning)

- Text Data의 의미가 전혀 알려져 있지 않고 기계학습을 통해 Text Data의 의미를 추정하는 경우.
- PCA, Cluster Analysis
- 연구자가 분석 단계마다 자신의 주관적 판단의 개입이 필요.

# 기존 Data Analysis vs Big Data Analysis

기존 Data Analysis	Big Data Analysis
<ul style="list-style-type: none"><li>• Data Mining</li><li>• Machine Learning</li></ul>	<ul style="list-style-type: none"><li>• Text Mining</li><li>• Sentiment Analysis</li><li>• Social Network Analysis</li><li>• Text Clustering</li></ul>

# 자연어 처리

- 자연어 처리 (Natural Language Processing, NLP)
  - 자연어 : 사람들이 일상적으로 사용하는 언어
  - 인공어 : Artificial Language, 사람들이 필요에 의해서 만든 언어  
에스페란토, C언어, Java 등

- 자연어 처리 분야
  - 자연어 이해 : 형태소 분석 → 의미 분석 → 대화 분석



- 자연어 처리의 활용 분야

- 맞춤법 검사, 번역기, 검색 엔진, 키워드 분석 등
- 문서 (Document) → 문단 (Paragraph) → 문장 (Sentence) → 어절 (Word phrase) → 형태소 (Morpheme) → 음절 (Syllable) → 음소 (Phoneme)

# 자연어 처리 용어

용어	상세
형태소(Morpheme)	의미를 가진 최소 단위입니다.
용언	꾸미는 말(동사, 형용사)입니다. 용언은 어근 + 어미로 구성됩니다.
어근(Stem)	용언이 활용할 때, 원칙적으로 모양이 변하지 않는 부분입니다.
어미	용언이 활용할 때, 변하는 부분으로 문법적 기능 수행합니다. 어미에는 연결 어미, 선어말 어미 + 종결 어미가 있습니다.
자모	문자 체계의 한 요소(자음, 모음)입니다.
품사	명사, 대명사, 수사, 동사, 형용사, 관형사, 부사, 감탄사, 조사가 있습니다.
어절 분류	명사 + 주격 조사, 명사 + 목적격 조사, 명사 + 관형격 조사, 동사 + 연결 어미 또는 동사 + 선어말 어미 + 종결 어미 등으로 분류합니다.
불용어(Stopword)	검색 등에서 의미가 없어 무시되도록 설정된 단어들입니다.
n-gram	문자의 빈도와 문자간 관계를 의미합니다. "안녕하세요"를 2-gram으로 나누면, "안녕", "녕하", "하세", "세요"로 나눌 수 있습니다.

# 자연어 처리 학습 주제

- 텍스트 전처리 : 토큰화, 정제, 형태소 분석, 불용어 처리.  
Label Encoding
- 개수 기반 단어 표현 : 문장 내에서 단어들의 빈도수를 측정해서 이를 기반으로 데이터를 분석할 수 있는 형태로 만드는 것
- 문서 유사도(Document Similarity) : 단어들을 수치화 한 후 이를 기반으로 단어들 사이의 거리를 계산해서 문서 간의 단어들의 차이를 계산하는 것
- Topic Modeling : 텍스트 본문의 숨겨진 의미 구조를 발견하기 위해 사용되는 Text Mining 기법
- 연관 분석(Association) : 문서 내의 단어들을 이용해서 연관 분석을 실시

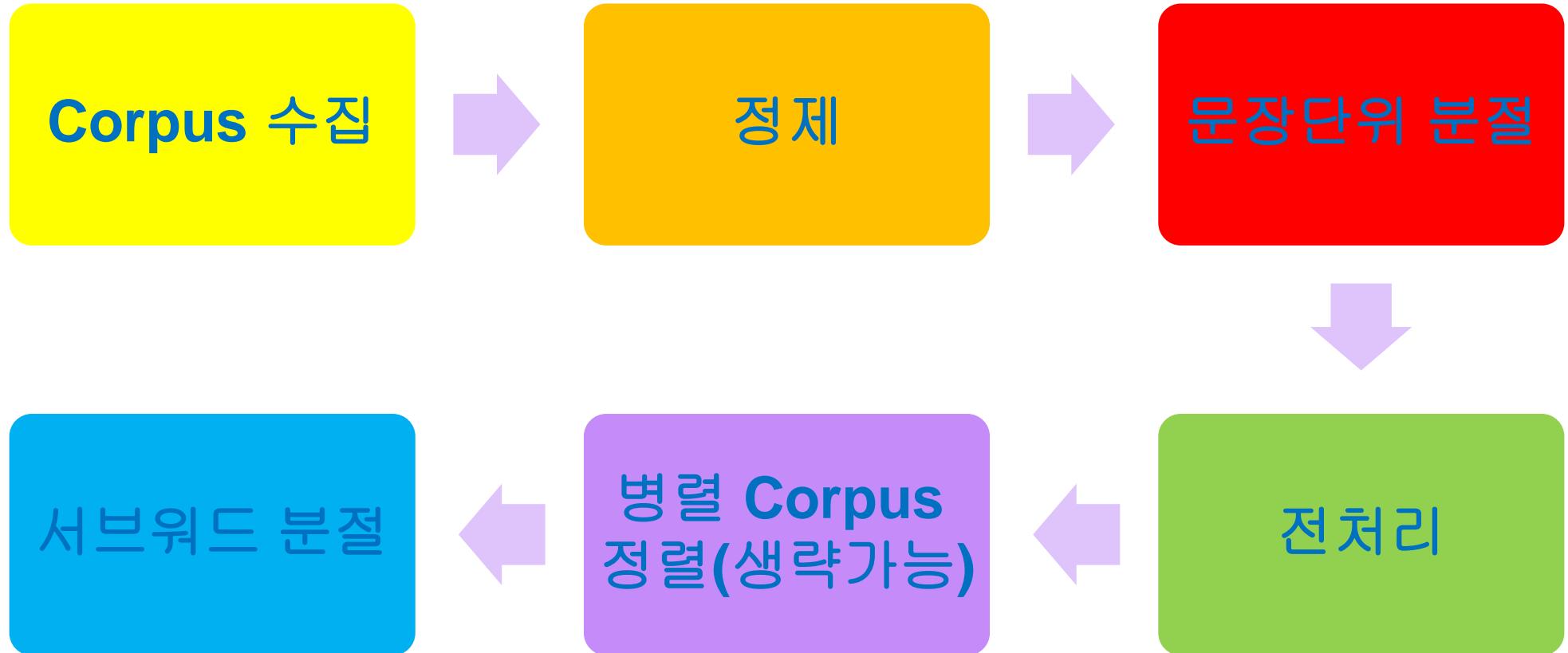
# 자연어 처리 학습 주제 (Cont.)

- Deep Learning을 이용한 자연어 처리 : RNN, LSTM 등의 인공 신경망 Algorithm을 이용해서 Deep Learning으로 자연어 처리
- Word Embedding : Word2vec 패키지를 이용해서 단어를 벡터로 표현하는 방법으로 희소 표현에서 밀집 표현으로 변환하는 것
- 텍스트 분류(Text Classification) : 텍스트를 입력으로 받아, 텍스트가 어떤 종류의 범주(Class)에 속하는지를 구분하는 작업
- Tagging : 각 단어가 어떤 유형에 속해 있는지를 알아내는 것
- 번역(Translation) : 챗봇(Chatbot) 또는 기계 번역(Machine Translation)

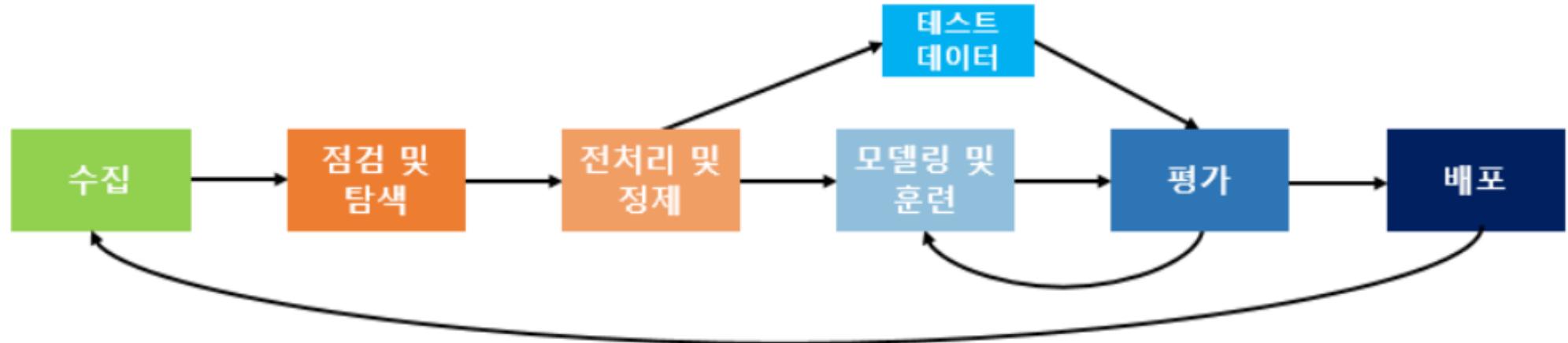
# 자연어(Natural Language Processing, NLP)

- 사람이 사용하는 언어를 Computer로 처리하는 각종 방법
- 크게 2가지로 나눈다
  - 통계적 자연어 처리
  - 언어학 자연어 처리

# 자연어(Natural Language Processing, NLP) Workflow



# Machine Learning Workflow



# Corpus 수집

- 공개된 Data의 Crawling or 구매
- txt, csv, xml, html, Database ...
- urllib.request, requests, BeautifulSoup, lxml, Selenium, Scrapy ...
- 저작권 같은 법적인 문제 조심
- 불필요한 Traffic으로 Web Server 부담 가중

# 점검 및 탐색

- 탐색적 데이터 분석(Exploratory Data Analysis, EDA) 단계
- 수집된 Data를 점검하고 탐색하는 단계
- 대소문자 통일
- 정규 표현식을 이용한 정제
- Noise Data, Data의 구조 파악 등.

# 문장 단위 분절

- 한 개의 line에는 한 문장만 있어야
- 단순한 마침표 기준으로 처리시 문제점 발생(예:U.S.)
- Algorithm 필요 or NLTK(>=3.2.5)

자연어처리는 인공지능의 한 출판입니다. seq2seq의 등장 이후로 Deep Learning을 활용한 자연어처리는 새로운 전기를 맞이하게 되었습니다. 문장을 받아 단순히 수치로 나타내던 시절을 넘어, 원하는 대로 문장을 만들어낼 수 있게 된 것입니다.

자연어처리는 인공지능의 한 출판입니다. seq2seq의 등장 이후로 \n Deep Learning을 활용한 자연어처리는 새로운 전기를 맞이하게 \n 되었습니다. 문장을 받아 단순히 수치로 나타내던 시절을 넘어, 원하는 \n 대로 문장을 만들어낼 수 있게 된 것입니다.

# 전처리 및 정제(Preprocessing and Cleaning)

- Tokenizing
- Cleaning
- Normalization
- Stemming
- Lemmatizing
- Stopword Extraction
- Splitting Data
- Integer Encoding & One-Hot Encoding
- Subword Segmentation

# 전처리 및 정제(Preprocessing and Cleaning) (Cont.)

언어	프로그램명	제작 언어	특징
한국어	Mecab	C++	일본어 Mecab을 wrappin했으며, 속도가 가장 빠름. Windows에서 사용 불가, 설치 다소 까다로움.
한국어	KoNLPy	Python Wrapping	PIP를 통해 설치 가능. 사용이 쉬우나 속도가 다소 느림.
일본어	Mecab	C++	속도가 빠름.
중국어	Stanford Parser	Java	미국 Standford에서 개발
중국어	PKU Parser	Java	북경대에서 개발. 위와 거의 성능 차이 없음.
중국어	Jieba	Python	가장 최근에 개발. Python으로 개발되어 시스템 구성에 용이.

- KorQuAD(<https://korquad.github.io/>)
- Khaiii(<https://tech.kakao.com/2018/12/13/khaiii/>)

# Subword 분절

- BPE(Byte Pair Encoding) Algorithm 사용
- 단어는 의미를 가진 더 작은 서브워드들의 조합으로 이루어 진다는 가정으로 사용
- 어휘 수를 줄이고, 효과적으로 희소성 줄어듦.

언어	단어	조합
영어	concentrate	con(=together) + centr(=center) + ate(=make)
한국어	집중(集中)	集(모을 집) + 中(가운데 중)

# Subword 분절 (Cont.)

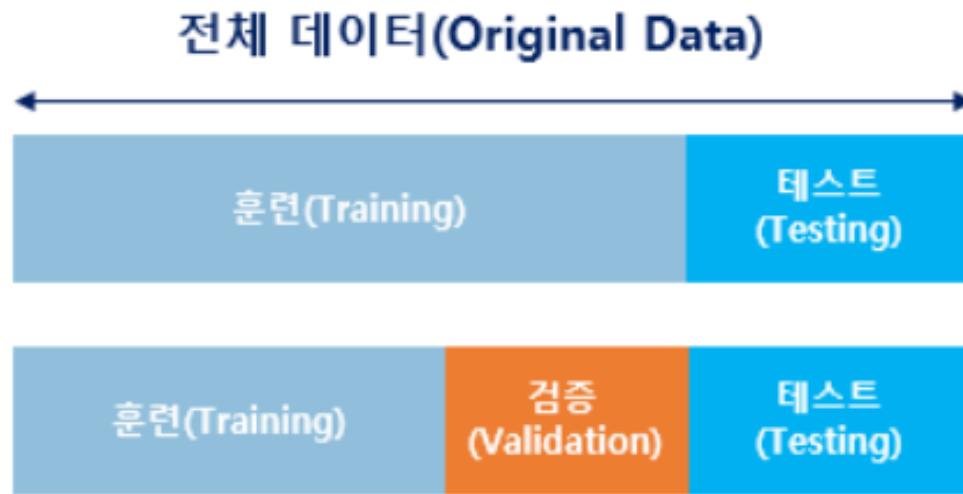
언어	단어	조합
영어	seq2seq streams	se + q + 2 + se + q stream + s
한국어	러닝 신경망 자연어	러 + 닝 신경 + 망 자연 + 어

## ■ Open Source

- Sennrich
  - <https://github.com/rsennrich/subword-nmt>
- Google의 SentencePiece
  - <https://github.com/google/sentencepiece>

# Modeling and Training

- 적절한 Machine Learning Algorithm 선택
- 전처리가 완료된 Data(학습지)를 가지고 기계에게 학습
- Test Data(수능시험) 준비
- 검증(Validation) Data(모의고사)



# 평가(Evaluation)와 배포(Deployment)

- 훈련 후, 학습된 Data를 Test Data로 평가
- 다양한 Machine Learning Algorithm에 따라 다양한 평가방법 사용
- 학습이 성공적이면 완성된 Model 배포

# NLTK 자연어 처리 Package

# NLTK(Natural Language Toolkit) Package

- 교육용으로 개발된 자연어 처리와 문서 분석용 Python Package
- <https://www.nltk.org>
- NLTK 패키지가 제공하는 주요 기능
  - 말뭉치(corpus)
  - 토큰 생성(tokenizing)
  - 형태소 분석(morphological analysis)
  - 품사 태깅(POS tagging)
- Cross-Platform Open Source

# NLTK(Natural Language Toolkit) Package (Cont.)

언어 처리 분야	NLTK Module	기능
Corpora 접근	corpus	corpora와 lexicon에 대한 표준화된 interface
문자열 처리	tokenize, stem	Tokenizing, 어간 추출(Stemming)
Collocation 발견	collocations	t-test, chi-squared, point-wise
품사 Tagging	tag	n-gram, backoff, Brill, HMM, TnT
기계 학습	classify, cluster, tbl	Decision Making Tree, Naïve Bayes, k-means 등
Chunking	chunk	정규식, n-gram, named-entity
Parsing	parse, ccg	Chart, 특징 기반, 단일화, 확률론적 의존성
구문 해석	sem, inference	Lambda 미적분학, 1차 논리, 모델 검사
평가 지표	metrics	정밀도(precision), 재현율(recall), 계수 일치도 (Agreement coefficients)
확률 및 추정	probability	빈도 분포, 확률 분포
응용프로그램	app, chat	용어 색인기(Concordancer), Parser, WorldNet Browser, Chatbot
언어 현장 조사	toolbox	SIL, Toolbox 형식으로 데이터 조작

# 말뭉치(Corpus)

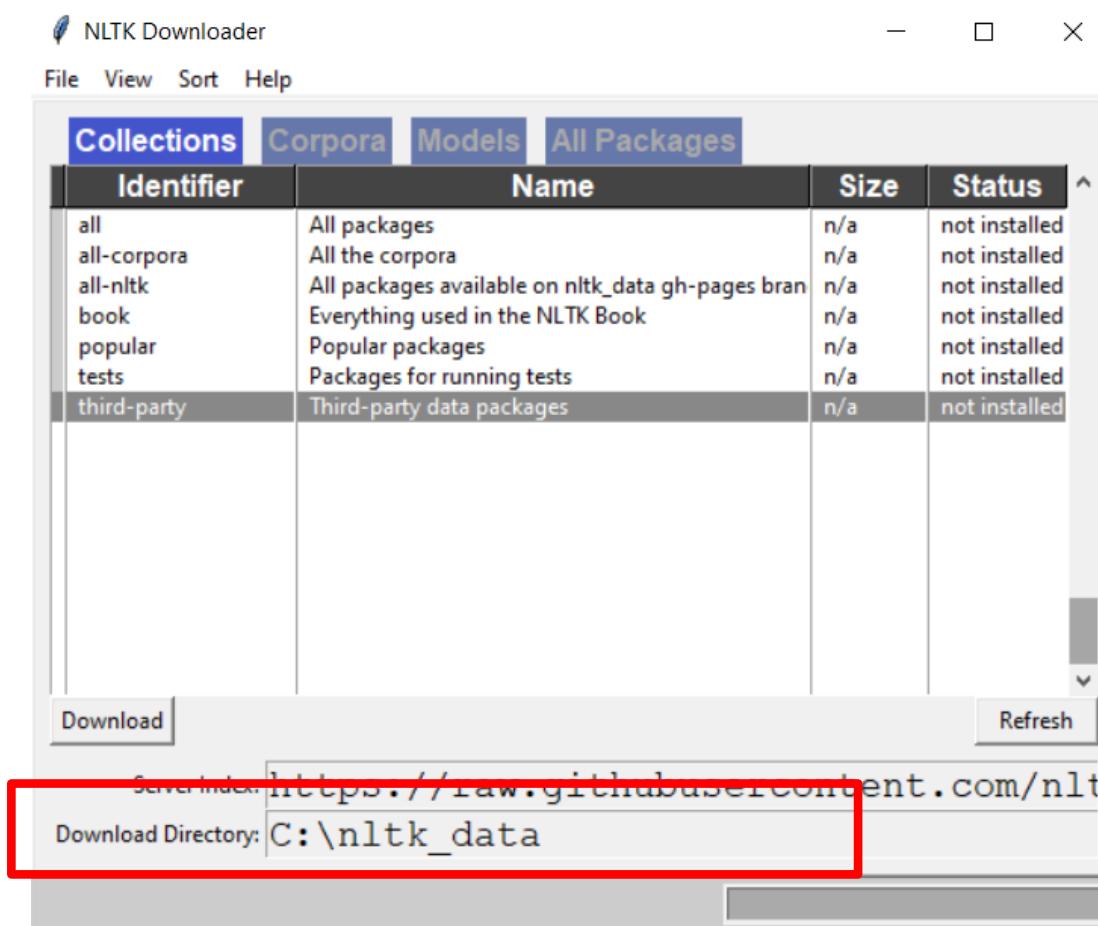
- 연구 대상 분야의 언어 현실을 총체적으로 보여 주는 자료의 집합
- Computer를 이용해 자연어 분석 작업을 할 수 있도록 만든 문서 집합
- Machine Learning을 수행하기 위한 훈련 Data
- NLTK에서는
  - 단순히 소설, 신문 등의 문서
  - 품사, 형태소 등의 보조적 의미를 추가
  - 쉬운 분석을 위해 구조적인 형태로 정리
- NLTK에서 corpus module로 학습용 말뭉치 제공
- <http://www.nltk.org/data.html>
- `nltk.download()` 함수로 download 해야 함.

# 말뭉치(Corpus) (Cont.)

```
import nltk
```

```
nltk.download()
```

showing info [https://raw.githubusercontent.com/nltk/nltk\\_data/gh-pages/index.xml](https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml)



# Penn Treebank Data

- NLTK를 설명할 목적으로 사용
- Tokenizing, Tagging, Chunking, Parsing 등에 사용
- Treebank
  - 구문 주석 말뭉치
  - 구조 분석을 좀 더 정교하게 만든 Tree Structure의 집합
- 400만 어절 규모의 Treebank 중 가장 유명한 것
- 주로 WallStreet Journal의 문장들로 구성.
- **nltk.download('treebank')**

```
[nltk_data] Downloading package treebank to C:\nltk_data...
[nltk_data] Package treebank is already up-to-date!
```

# Penn Treebank Data (Cont.)

NLTK Downloader

File View Sort Help

Collections Corpora Models All Packages

Identifier	Name	Size	Status
shakespeare	Shakespeare XML Corpus Sample	464.3 KB	not installed
sinica_treebank	Sinica Treebank Corpus Sample	878.2 KB	not installed
smultron	SMULTRON Corpus Sample	162.3 KB	not installed
state_union	C-Span State of the Union Address Corpus	789.8 KB	not installed
stopwords	Stopwords Corpus	22.6 KB	not installed
subjectivity	Subjectivity Dataset v1.0	509.4 KB	not installed
swadesh	Swadesh Wordlists	22.3 KB	not installed
switchboard	Switchboard Corpus Sample	772.6 KB	not installed
timit	TIMIT Corpus Sample	21.2 MB	not installed
toolbox	Toolbox Sample Files	244.7 KB	not installed
treebank	Penn Treebank Sample	1.7 MB	installed
twitter_samples	Twitter Samples	15.3 MB	not installed
udhr	Universal Declaration of Human Rights Corpus	1.1 MB	not installed
udhr2	Universal Declaration of Human Rights Corpus (Ur	1.6 MB	not installed
unicode_samples	Unicode Samples	1.2 KB	not installed
universal_treebanks_v	Universal Treebanks Version 2.0	24.7 MB	not installed

Download Refresh

Server Index: [https://raw.githubusercontent.com/nltk/nltk\\_data/3.2.5/corpora/treebank.zip](https://raw.githubusercontent.com/nltk/nltk_data/3.2.5/corpora/treebank.zip)

Download Directory: C:\nltk\_data

Finished installing treebank

# Penn Treebank Data (Cont.)

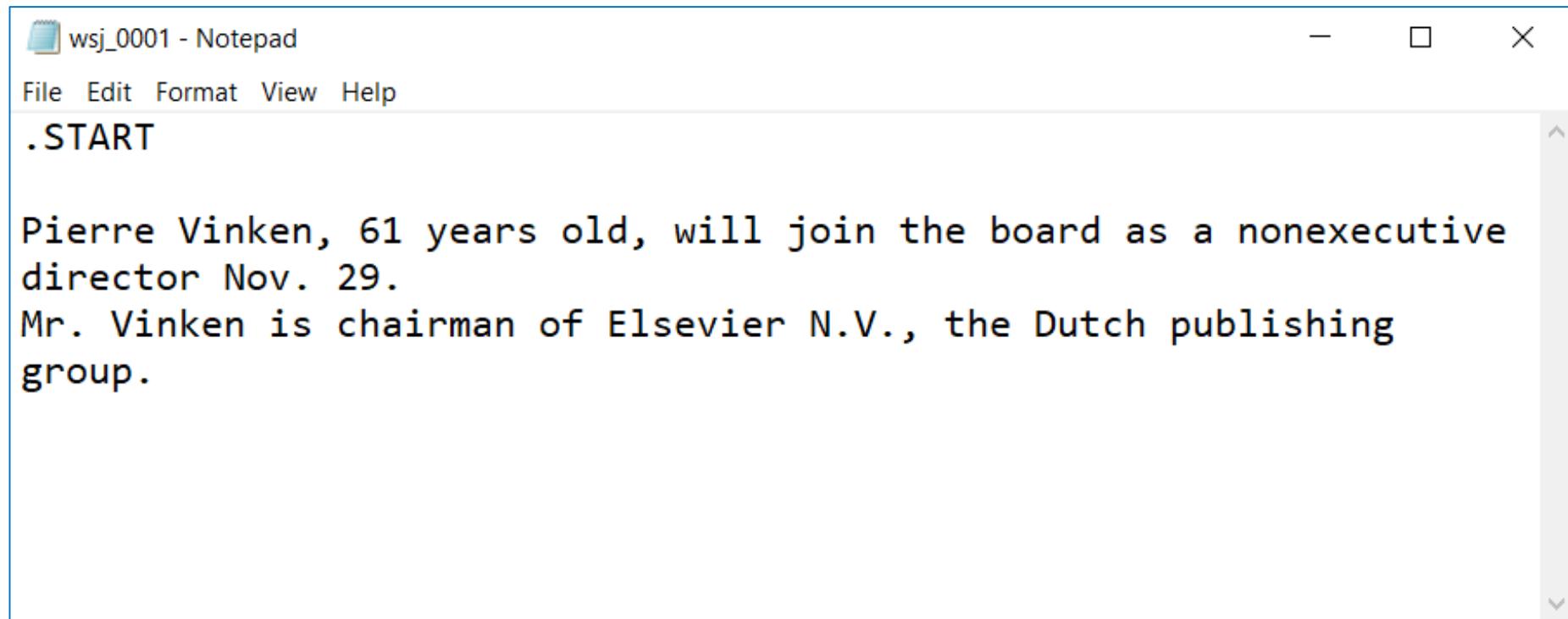
## ■ File들의 id 확인

```
from nltk.corpus import treebank  
print(treebank.fields())
```

```
['wsj_0001.mrg', 'wsj_0002.mrg', 'wsj_0003.mrg', 'wsj_0004.mrg', 'wsj_0005.mrg', 'wsj_0006.mrg', 'wsj_0007.mrg', 'wsj_0008.mrg', 'wsj_0009.mrg', 'wsj_0010.mrg', 'wsj_0011.mrg', 'wsj_0012.mrg', 'wsj_0013.mrg', 'wsj_0014.mrg', 'wsj_0015.mrg', 'wsj_0016.mrg', 'wsj_0017.mrg', 'wsj_0018.mrg', 'wsj_0019.mrg', 'wsj_0020.mrg', 'wsj_0021.mrg', 'wsj_0022.mrg', 'wsj_0023.mrg', 'wsj_0024.mrg', 'wsj_0025.mrg', 'wsj_0026.mrg', 'wsj_0027.mrg', 'wsj_0028.mrg', 'wsj_0029.mrg', 'wsj_0030.mrg', 'wsj_0031.mrg', 'wsj_0032.mrg', 'wsj_0033.mrg', 'wsj_0034.mrg', 'wsj_0035.mrg', 'wsj_0036.mrg', 'wsj_0037.mrg', 'wsj_0038.mrg', 'wsj_0039.mrg', 'wsj_0040.mrg', 'wsj_0041.mrg', 'wsj_0042.mrg', 'wsj_0043.mrg', 'wsj_0044.mrg', 'wsj_0045.mrg', 'wsj_0046.mrg', 'wsj_0047.mrg', 'wsj_0048.mrg', 'wsj_0049.mrg', 'wsj_0050.mrg', 'wsj_0051.mrg', 'wsj_0052.mrg', 'wsj_0053.mrg', 'wsj_0054.mrg', 'wsj_0055.mrg', 'wsj_0056.mrg', 'wsj_0057.mrg', 'wsj_0058.mrg', 'wsj_0059.mrg', 'wsj_0060.mrg', 'wsj_0061.mrg', 'wsj_0062.mrg', 'wsj_0063.mrg', 'wsj_0064.mrg', 'wsj_0065.mrg', 'wsj_0066.mrg', 'wsj_0067.mrg', 'wsj_0068.mrg', 'wsj_0069.mrg', 'wsj_0070.mrg', 'wsj_0071.mrg', 'wsj_0072.mrg', 'wsj_0073.mrg', 'wsj_0074.mrg', 'wsj_0075.mrg', 'wsj_0076.mrg', 'wsj_0077.mrg', 'wsj_0078.mrg', 'wsj_0079.mrg', 'wsj_0080.mrg', 'wsj_0081.mrg', 'wsj_0082.mrg', 'wsj_0083.mrg', 'wsj_0084.mrg', 'wsj_0085.mrg', 'wsj_0086.mrg', 'wsj_0087.mrg', 'wsj_0088.mrg', 'wsj_0089.mrg', 'wsj_0090.mrg', 'wsj_0091.mrg', 'wsj_0092.mrg', 'wsj_0093.mrg', 'wsj_0094.mrg', 'wsj_0095.mrg', 'wsj_0096.mrg', 'wsj_0097.mrg', 'wsj_0098.mrg', 'wsj_0099.mrg', 'wsj_0100.mrg', 'wsj_0101.mrg', 'wsj_0102.mrg', 'wsj_0103.mrg', 'wsj_0104.mrg', 'wsj_0105.mrg', 'wsj_0106.mrg', 'wsj_0107.mrg', 'wsj_0108.mrg', 'wsj_0109.mrg', 'wsj_0110.mrg', 'wsj_0111.mrg', 'wsj_0112.mrg', 'wsj_0113.mrg', 'wsj_0114.mrg', 'wsj_0115.mrg', 'wsj_0116.mrg', 'wsj_0117.mrg', 'wsj_0118.mrg', 'wsj_0119.mrg', 'wsj_0120.mrg', 'wsj_0121.mrg', 'wsj_0122.mrg', 'wsj_0123.mrg', 'wsj_0124.mrg', 'wsj_0125.mrg', 'wsj_0126.mrg', 'wsj_0127.mrg', 'wsj_0128.mrg', 'wsj_0129.mrg', 'wsj_0130.mrg', 'wsj_0131.mrg', 'wsj_0132.mrg', 'wsj_0133.mrg', 'wsj_0134.mrg', 'wsj_0135.mrg', 'wsj_0136.mrg', 'wsj_0137.mrg', 'wsj_0138.mrg', 'wsj_0139.mrg', 'wsj_0140.mrg', 'wsj_0141.mrg', 'wsj_0142.mrg', 'wsj_0143.mrg', 'wsj_0144.mrg', 'wsj_0145.mrg', 'wsj_0146.mrg', 'wsj_0147.mrg', 'wsj_0148.mrg', 'wsj_0149.mrg', 'wsj_0150.mrg', 'wsj_0151.mrg', 'wsj_0152.mrg', 'wsj_0153.mrg', 'wsj_0154.mrg', 'wsj_0155.mrg', 'wsj_0156.mrg', 'wsj_0157.mrg', 'wsj_0158.mrg', 'wsj_0159.mrg', 'wsj_0160.mrg', 'wsj_0161.mrg', 'wsj_0162.mrg', 'wsj_0163.mrg', 'wsj_0164.mrg', 'wsj_0165.mrg', 'wsj_0166.mrg', 'wsj_0167.mrg', 'wsj_0168.mrg', 'wsj_0169.mrg', 'wsj_0170.mrg', 'wsj_0171.mrg', 'wsj_0172.mrg', 'wsj_0173.mrg', 'wsj_0174.mrg', 'wsj_0175.mrg', 'wsj_0176.mrg', 'wsj_0177.mrg', 'wsj_0178.mrg', 'wsj_0179.mrg', 'wsj_0180.mrg', 'wsj_0181.mrg', 'wsj_0182.mrg', 'wsj_0183.mrg', 'wsj_0184.mrg', 'wsj_0185.mrg', 'wsj_0186.mrg', 'wsj_0187.mrg', 'wsj_0188.mrg', 'wsj_0189.mrg', 'wsj_0190.mrg', 'wsj_0191.mrg', 'wsj_0192.mrg', 'wsj_0193.mrg', 'wsj_0194.mrg', 'wsj_0195.mrg', 'wsj_0196.mrg', 'wsj_0197.mrg', 'wsj_0198.mrg', 'wsj_0199.mrg']
```

# Penn Treebank Data (Cont.)

C:\nltk\_data\corpora\treebank\raw\wjs\_0001



The screenshot shows a Windows Notepad window titled "wsj\_0001 - Notepad". The window contains the following text:

```
.START

Pierre Vinken, 61 years old, will join the board as a nonexecutive
director Nov. 29.
Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing
group.
```

# Penn Treebank Data (Cont.)

- Tokenizing
- **sents()** 이용
- 문장의 token化한 결과

```
1 treebank.sents('wsj_0001.mrg')
```

```
[['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'No  
v.', '29', '.'], ['Mr.', 'Vinken', 'is', 'chairman', 'of', 'Elsevier', 'N.Y.', ',', 'the', 'Dutch', 'publishing', 'group', '.']]
```

# Penn Treebank Data (Cont.)

- 원문 보기
- **join()**

```
1 wsj_0001 = treebank.sents('wsj_0001.mrg')
2 for line in wsj_0001:
3     print(''.join(line))
```

Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .  
Mr. Vinken is chairman of Elsevier N.Y. , the Dutch publishing group .

# Penn Treebank Data (Cont.)

- 품사 Tagging
- tagged\_words()

```
1 treebank.tagged_words('wsj_0001.mrg')  
[('Pierre', 'NNP'), ('Vinken', 'NNP'), ('.', '.', '.', '.'), ...]
```

# Penn Treebank Data (Cont.)

- Data parsing 및 각 단어들의 상세 tagging
- **parsed\_sents()**

```
1 treebank.parsed_sents('wsj_0001.mrg')[0]
```

The Ghostscript executable isn't found.  
See <http://web.mit.edu/ghostscript/www/Install.htm>

If you're using a Mac, you can try installing  
<https://docs.brew.sh/Installation> then `brew install ghostscript`

```
Tree('S', [Tree('NP-SBJ', [Tree('NP', [Tree('NNP', ['Pierre']), Tree('NNP', ['Vinken'])]), Tree(',', ['.', '.']), Tree('ADJP', [Tree('NP', [Tree('CD', ['61']), Tree('NNS', ['years'])]), Tree('JJ', ['old'])]), Tree(',', ['.', '.']), Tree('VP', [Tree('MD', ['will']), Tree('VP', [Tree('VB', ['join']), Tree('NP', [Tree('DT', ['the']), Tree('NN', ['board'])]), Tree('PP-CLR', [Tree('IN', ['as']), Tree('NP', [Tree('DT', ['a']), Tree('JJ', ['nonexecutive'])]), Tree('NN', ['director'])])]), Tree('NP-TMP', [Tree('NNP', ['Nov.']), Tree('CD', ['29'])])]), Tree('.', ['.'])])])])
```

# book Data

- `nltk.download('book')`
- NLTK 책에서 사용하는 모든 Data downloads.

```
import nltk  
nltk.download('book', quiet=True)  
from nltk.book import *
```

```
1 import nltk  
2 nltk.download('book', quiet=True)  
True  
  
1 from nltk.book import *  
*** Introductory Examples for the NLTK Book ***  
Loading text1, ..., text9 and sent1, ..., sent9.  
Type the name of the text or sentence to view it.  
Type: 'texts()' or 'sents()' to list the materials.  
text1: Moby Dick by Herman Melville 1851  
text2: Sense and Sensibility by Jane Austen 1811  
text3: The Book of Genesis  
text4: Inaugural Address Corpus  
text5: Chat Corpus  
text6: Monty Python and the Holy Grail  
text7: Wall Street Journal  
text8: Personalis Corpus  
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

# book Data (Cont.)

- 허먼 멜빌(Herman Melville, 1819.8.1 ~ 1891.9.28, 미국)의 소설 Moby Dick

```
1 type(text1)
```

```
nltk.text.Text
```

```
1 text1
```

```
<Text : Moby Dick by Herman Melville 1851>
```

# Gutenberg Data

- nltk의 corpus module 중 parsing이 안된 원문 데이터
- Gutenberg 말뭉치는 저작권이 말소된 문학작품.

```
1 nltk.download('gutenberg')
```

```
[nltk_data] Downloading package gutenberg to C:\#nltk_data...
[nltk_data] Package gutenberg is already up-to-date!
```

True

```
1 nltk.corpus.gutenberg.fileids()
```

```
['austen-emma.txt',
 'austen-persuasion.txt',
 'austen-sense.txt',
 'bible-kjv.txt',
 'blake-poems.txt',
 'bryant-stories.txt',
 'burgess-busterbrown.txt',
 'carroll-alice.txt',
 'chesterton-ball.txt',
 'chesterton-brown.txt',
 'chesterton-thursday.txt',
 'edgeworth-parents.txt',
 'melville-moby_dick.txt',
 'milton-paradise.txt',
 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt',
 'shakespeare-macbeth.txt',
 'whitman-leaves.txt']
```

# Gutenberg Data (Cont.)

- 제인 오스틴(Jane Austen, 1775.12.16 ~ 1817.7.18, 영국의 소설가)의 소설 엠마(austen\_emma.txt)

```
1 emma = nltk.corpus.gutenberg.raw('austen-emma.txt')
2 emma[:289]
```

'[Emma by Jane Austen 1816] VOLUME I  
CHAPTER I  
Emma Woodhouse, handsome, clever, and rich, with a comfortable home and happy disposition, seemed to unite some of the best blessings of existence; and had lived nearly twenty-one years in the world without very little to distress or vex her.'

# Text Class

- 문서 분석에 유용한 Method 제공
- 먼저, 예제에 사용할 **emma** 데이터를 불러와 정규표현식을 이용해 토큰화 한 후 Text 객체로 만든다.

```
1 emma = nltk.corpus.gutenberg.raw("austen-emma.txt")
```

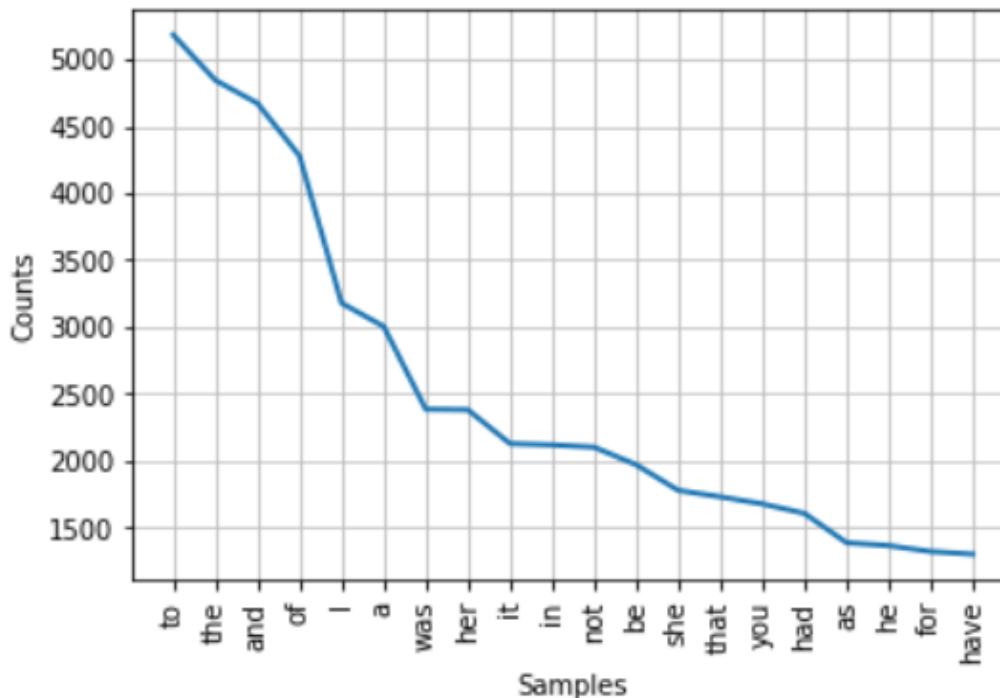
```
1 from nltk.tokenize import RegexpTokenizer  
2 retokenize = RegexpTokenizer("[\w]+")  
3 from nltk import Text  
4 emma_text = Text(retokenize.tokenize(emma), name="Emma")
```

# Text Class (Cont.)

## ■ plot()

- 각 단어(토큰)의 사용 빈도를 그래프로

```
1 import matplotlib.pyplot as plt  
2  
3 emma_text.plot(20)  
4 plt.show()
```



# Text Class (Cont.)

## ■ **concordance()**

- 해당 단어가 들어간 문장을 찾아준다.
- line 매개변수로 원하는 개수 지정 가능.

```
1 emma_text.concordance('Emma', lines=5)
```

Displaying 5 of 865 matches:

Emma by Jane Austen 1816 VOLUME I CHAPTER  
Jane Austen 1816 VOLUME I CHAPTER I Emma Woodhouse handsome clever and rich w  
f both daughters but particularly of Emma Between \_them\_ it was more the intim  
nd friend very mutually attached and Emma doing just what she liked highly est  
by her own The real evils indeed of Emma s situation were the power of having

# Text Class (Cont.)

## ■ **similar()**

- 문맥이 비슷한 단어를 찾아준다.

```
1 emma_text.similar("general")
```

love short it london time fact her which highbury and own well him  
kind return vain person despair law bath

```
1 emma_text.similar("general", 10)
```

love short it london time fact her which highbury and

```
1 emma_text.similar("strong")
```

good much large happy natural grateful interesting clever little long  
well soon great odd often right late sorry superior ready

# Text Class (Cont.)

## ■ common\_contexts()

- 둘 이상의 단어에 의해 공유되는 문맥을 조사한다.

```
1 | emma_text.common_contexts(["general", "strong"])
```

a\_expectation

# Text Class (Cont.)

## ■ **collocations\_list()**

- 연어(Collocation, 같이 붙어서 쓰이는 단어)를 찾는다.

```
1 print(emma_text.collocation_list())
```

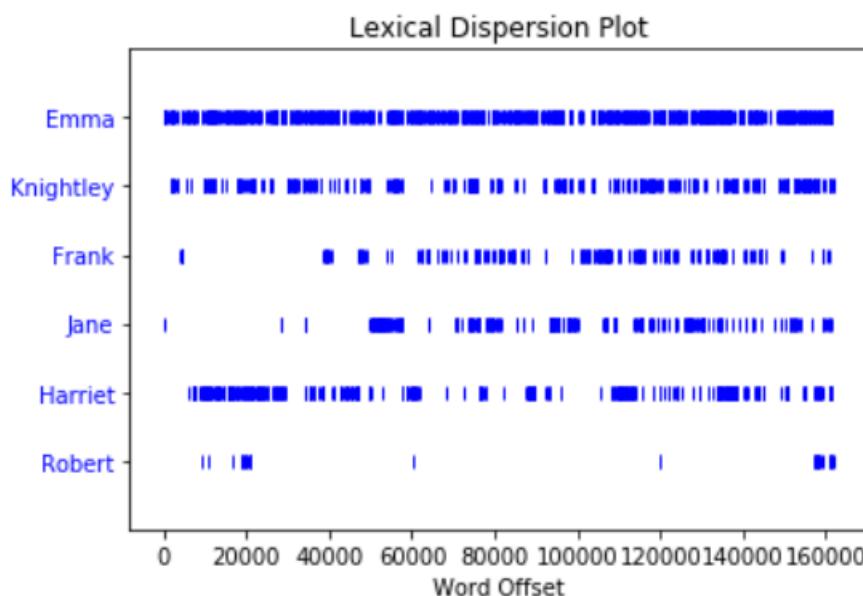
```
['Mrs Weston', 'Frank Churchill', 'Miss Woodhouse', 'Mrs Elton', 'Miss Bates', 'Jane Fairfax', 'Miss Fairfax', 'every thing', 'young man', 'every body', 'great deal', 'Mrs Goddard', 'dare say', 'Maple Grove', 'John Knightley', 'Miss Taylor', 'Miss Smith', 'Robert Martin', 'Colonel Campbell', 'Box Hill']
```

# Text Class (Cont.)

## ■ dispersion\_plot()

- 처음부터 지정한 단어가 얼마나 많이 나타날지 단어의 위치를 표시할 수 있다.
- 다음 예제에서는 소설 Emma의 각 등장인물들이 어느 시점에 언급되었는지 시각화 할 수 있다.
- 각 행은 전체 텍스트를 나타낸다.

```
1 emma_text.distribution_plot(["Emma", "Knightley", "Frank", "Jane", "Harriet", "Robert"])
```



# Text Class (Cont.)

## ■ len()

- 단어의 총 개수를 계산

```
1
```

```
len(emma_text)
```

```
161983
```

## ■ set()

- 중복을 제거한 단어의 수

```
1
```

```
len(set(emma_text))
```

```
7723
```

- 텍스트의 어휘 풍부성에 대한 척도를 계산할 수 있는데, 다음 예제에서 중복을 제거한 단어의 수가 전체 단어 수의 4.7%에 불과하다는 것을 보여준다.

```
1
```

```
len(set(emma_text)) / len(emma_text)
```

```
0.047677842736583466
```

# FreqDist Class

- 문서에 사용된 단어(토큰)의 사용빈도 정보를 담는 Class
- Text Class의 **vocab()**로 객체를 만들 수 있다.

```
1 import nltk  
2 emma = nltk.corpus.gutenberg.raw('austen-emma.txt')
```

```
1 from nltk.tokenize import RegexpTokenizer  
2 retokenize = RegexpTokenizer('[#w]+')  
3 from nltk import Text  
4 emma_text = Text(retokenize.tokenize(emma), name = 'Emma')
```

```
1 emma_fd = emma_text.vocab()  
2 type(emma_fd)
```

nltk.probability.FreqDist

# FreqDist Class (Cont.)

- vocab()를 사용하지 않고 다음처럼 단어 List를 이용해 직업 만들 수도 있다.
- 다음 코드에서는 Emma 말뭉치에서 사람의 이름만 모아서 FreqDist Class를 만들었다.
- 품사 태그에서 NNP(고유대명사)이면서 필요 없는 단어(stop words)는 제거했다.

```
1 from nltk.tag import pos_tag
2 from nltk import FreqDist
3 stopwords = ["Mr.", "Mrs.", "Miss", "Mr", "Mrs", "Dear"]
4 emma_tokens = pos_tag(emma_text)
5 names_list = [t[0] for t in emma_tokens if t[1] == "NNP" and t[0] not in stopwords]
6 emma_fd_names = FreqDist(names_list)
7 emma_fd_names

FreqDist({'Emma': 830, 'Harriet': 491, 'Weston': 439, 'Knightley': 389, 'Elton': 385, 'Woodhouse': 304, 'Jane': 299, 'Fairfax': 241, 'Churchill': 223, 'Frank': 208, ...})
```

# FreqDist Class (Cont.)

- FreqDist 클래스는 단어를 키(key), 출현빈도를 값(value)으로 가지는 dict 유형과 유사하다.
- 다음 코드는 전체 단어의 수, "Emma"라는 단어의 출현 횟수, 확률을 각각 출력한다.

```
1 emma_fd_names.N(), emma_fd_names["Emma"], emma_fd_names.freq("Emma")  
(7863, 830, 0.10555767518758744)
```

## ■ **most\_common()**

- 가장 출현 횟수가 높은 단어 찾기

```
1 emma_fd_names.most_common(5)  
[('Emma', 830),  
 ('Harriet', 491),  
 ('Weston', 439),  
 ('Knightley', 389),  
 ('Elton', 385)]
```

# 한글 형태소 분석



# 형태소

## ■ 형태소

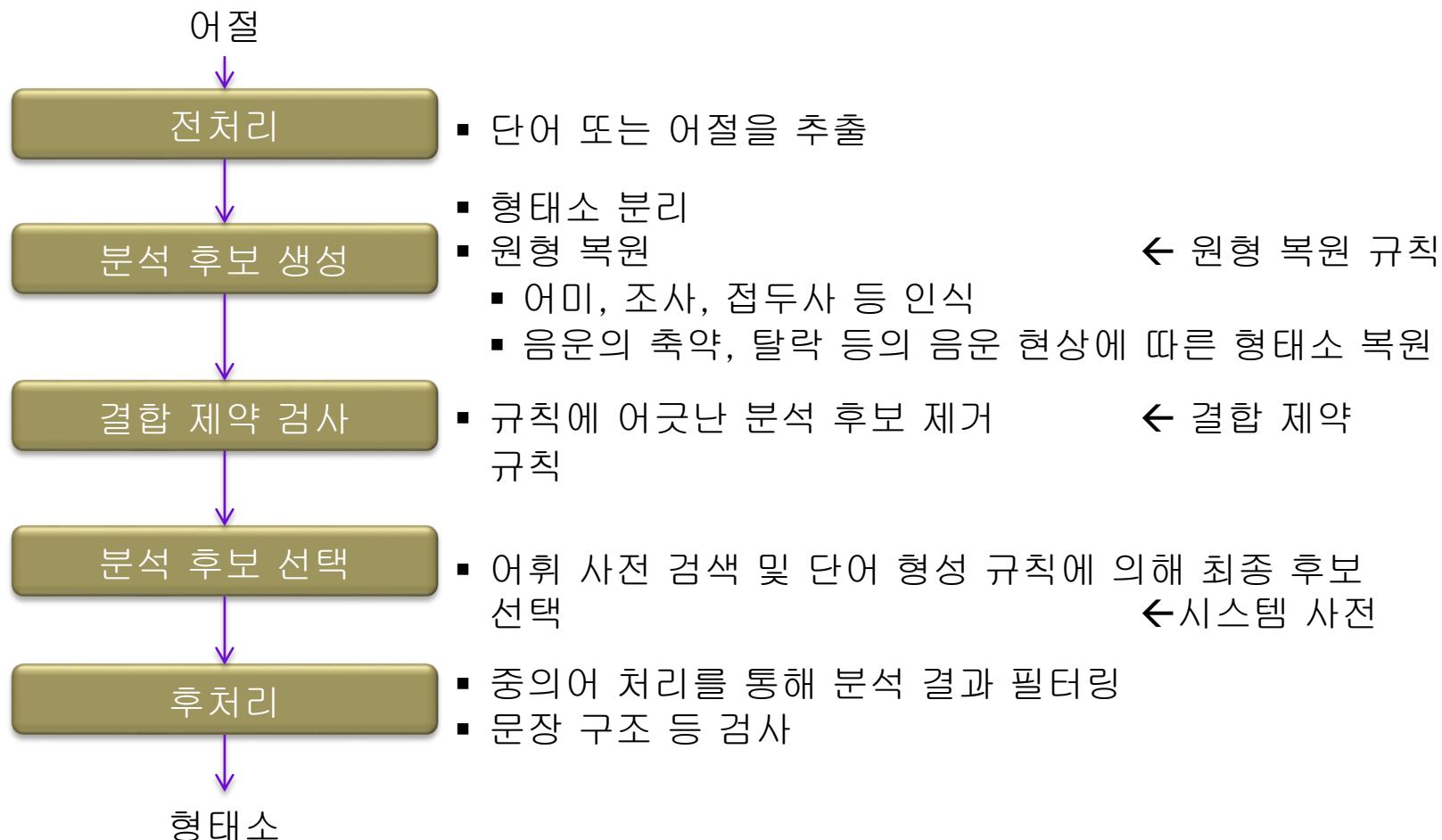
- 의미를 가진 최소의 단위
- 문법적, 관계적인 뜻을 나타내는 단어 또는 단어의 부분
- 체언 (명사, 대명사, 수사), 수식언 (관형사, 부사), 독립언 (감탄사), 관계언 (조사, 서술격조사),  
용언 (동사, 형용사, 조용보조어간), 부속어 (어미, 접미)

## ■ 형태소의 종류

- 실질 형태소
  - 체언, 수식언, 감탄사, 용언의 어근
  - 구체적인 대상, 동작, 상태 등을 나타내는 형태소로 어휘 형태소
  - 검색 엔진에서 색인의 대상이 됩니다.
- 형식 형태소
  - 조사, 어말어미, 접미사, 접두사, 선어말어미
  - 형태소 간의 관계를 나타내는 형태소로 문법 형태소

# 형태소 분석

- 단어 또는 어절을 구성하는 각 형태소 분리
- 분리된 형태소의 기본형(어근) 및 품사 정보 추출
- 일반적인 형태소 분석 절차



# 형태소 분석 엔진

## ■ KoNLPy

- Python용 형태소 분석기(Korean NLP in Python)(GPL v3+ License)
- 내부적으로 각기 다른 언어(Java, C++ 등)로 호환성에 문제가 생기기도
- 일부 Library 오래 전 제작/업데이트
- Java로 구현되어 있어서 C++로 구현된 Mecab에 속도에서 불리.
- 사용하기 용이하고 다양한 Library들 사용의 장점도.
- <http://konlpy.readthedocs.org/>

## ■ Mecab

- 한국어 형태소 분석에서 가장 많이 사용
- 원래 일본어 형태소 분석용 Open Source
- <https://bitbucket.org/eunjeon/mecab-ko-dic/src/master/>

## ■ KOMORAN

- Java로 만든 오픈소스 형태소 분석기(Apache v2 License)
- <https://shineware.tistory.com/tag/KOMORAN/>

# 형태소 분석 엔진 (Cont.)

## ■ HanNanum

- Java로 만들어진 형태소 분석기(GPL v3 License)
- <http://semanticweb.kaist.ac.kr/home/index.php/HanNanum>

## ■ KoNLP

- R용 형태소 분석기(Korean NLP)(GPL v3 License)
- <https://github.com/haven-jeon/KoNLP>

## ■ Khaiii

- Kakao Hangul Analyzer III
- 세종 Corpus를 이용해서 CNN 기술을 적용한 형태소 분석기
- C++/Python으로 구현
- <https://tech.kakao.com/2018/12/13/khaiii/>

# KoNLPy Package

- 한국어 정보처리를 위한 파이썬 패키지
- Open Source Software이며 [GPL v3 이상] 라이센스로 배포
- 공식 사이트는 다음과 같습니다.
  - <http://konlpy.org/en/latest/>
  - <https://github.com/konlpy/konlpy>
- KoNLPy 패키지 설치
  - KoNLPy는 JPype1 패키지에 의존
  - (base) C:\Users\COM> pip install konlpy
- JDK를 설치해야 함
  - <http://jdk.java.net/12/> -> Builds -> Windows/x64 zip 다운로드
  - 압축 풀기
  - JAVA\_HOME 환경변수 설정

# KoNLPy Package (Cont.)

- API
  - <https://konlpy-ko.readthedocs.io/ko/v0.4.3/api/konlpy.tag/>
- Hannum Class
- Kkma Class
- Komoran Class
- Mecab Class
- Okt(Open Korean Text, Twitter가 0.5.0부터 Okt로 변경됨) Class
- Korean POS tags comparison chart
  - <https://docs.google.com/spreadsheets/d/1OGAjUvalBuX-oZvZ-9tEfYD2gQe7hTGsgUpiiBSXI8/edit#gid=0>

# 형태소 분석

- 형태소를 비롯하여, 어근, 접두사/접미사, 품사(POS, part-of-speech) 등 다양한 언어적 속성의 구조를 파악하는 것
- 품사 태그를 알아야 함
  - <https://konlpy-ko.readthedocs.io/ko/v0.4.3/morph/#comparison-between-pos-tagging-classes>

# 형태소 관련 용어 번역

영어	한국어	예시
adverb	부사	매우
adjective	형용사	예쁘다
conjunction	접속사	그리고
determiner/prenoun	관형사	새-
noun	명사	아이유
pronoun	대명사	그녀
verb	동사	달리다
exclamations/interjections	감탄사	오!
particle/postposition	조사	이/가
preposition	전치사 (영어)	between

# 품사 Tagging

- 형태소의 뜻과 문맥을 고려하여 그것에 Markup을 하는 일
- 단어와 /(슬래시) 뒤에 품사가 표시되어 나누어지도록 하는 것
- 예)
  - 원문 : 가방에 들어가신다
  - 품사 태깅 : 가방/NNG + 에/JKM + 들어가/VV + 시/EPH + 냐다/EFN

# Mecab

- 한국어 분절에 가장 많이 사용.
- Yet Another Part-of-Speech and Morphological Analyzer
- 원래 일본어 형태소 분석용 Open Source였으나, 이를 한국어 형태소 분석기로 적용.
- <http://taku910.github.io/mecab/>
- API
  - <https://konlpy-ko.readthedocs.io/ko/v0.4.3/api/konlpy.tag/#mecab-class>

## Mecab (Cont.)

```
konlpy.tag._mecab.Mecab(dicpath='/usr/local/lib/mecab/dic/mecab-ko-dic')
```

메서드	설명
morphs(phrase)	형태소 분석 문구를 반환.
nouns(phrase)	명사를 추출.
pos(phrase, flatten=True)	flatten이 False이면 어절을 보존.

# mecab-ko-dic

- Open Source 형태소 분석 engine인 MeCab을 사용하여, 한국어 형태소 분석을 하기 위한 프로그램.
- Apache License v. 2.0
- <https://bitbucket.org/eunjeon/mecab-ko-dic/src/master/>
- mecab-ko-dic에서 사용한 사전형식과 품사 tag
  - <https://docs.google.com/spreadsheets/d/1-9blXKjtjeKZqsf4NzHeYJCrr49-nXeRF6D80udfcwY/edit#gid=589544265>

# mecab-ko-dic (Cont.)

```
instructor@MyDesktop:~$ echo "안녕하세요, 반갑습니다!" | mecab
안녕      NNG, 행위, T, 안녕, *, *, *, *
하          XSV, *, F, 하, *, *, *, *
세요      EP+EF, *, F, 세요, Inflect, EP, EF, 시 /EP/*+어 요 /EF/*
,          SC, *, *, *, *, *, *, *
반갑      VA, *, T, 반갑, *, *, *, *
습니다    EF, *, F, 습니다, *, *, *, *
!          SF, *, *, *, *, *, *, *
EOS
```

# Mecab (Cont.)

```
from konlpy.tag import Mecab  
mecab = Mecab()
```

```
text = u"""아름답지만 다소 복잡하기도한 한국어는 전세계에서 13번째로 많이 사용되는 언어입니다."""
```

```
mecab.morphs(text)
```

```
['아름답',  
'지만',  
'다소',  
'복잡',  
'하',  
'기',  
'도',  
'한',  
'한국어',  
'는',  
'전',  
'세계',  
'에서',  
'13',  
'번',  
'째',  
'로',  
'많이',  
'사용',  
'되',  
'는',  
'언어',  
'입니다',  
. . .]
```

# Mecab (Cont.)

```
mecab.nouns(text)
```

```
['한국어', '세계', '번', '사용', '언어']
```

```
mecab.pos(text)
```

```
[('아름답', 'VA'),
 ('지만', 'EC'),
 ('다소', 'MAG'),
 ('복잡', 'XR'),
 ('하', 'XSA'),
 ('기', 'ETN'),
 ('도', 'JX'),
 ('한', 'MM'),
 ('한국어', 'NNG'),
 ('는', 'JX'),
 ('전', 'MM'),
 ('세계', 'NNG'),
 ('에서', 'JKB'),
 ('13', 'SN'),
 ('번', 'NNBC'),
 ('째', 'XSN'),
 ('로', 'JKB'),
 ('많이', 'MAG'),
 ('사용', 'NNG'),
 ('되', 'XSV'),
 ('는', 'ETM'),
 ('언어', 'NNG'),
 ('입니다', 'VCP+EF'),
 ('.', 'SF')]
```

# Komoran

- 2013년 Shineware에서 Java로 만든 오픈소스 한국어 형태소 분석기

**konlpy.tag.Komoran(jvmpath=None,  
dicpath=None)**

메서드	설명
morphs(phrase)	형태소 분석 문구를 반환.
nouns(phrase)	명사를 추출.
pos(phrase, flatten=True)	flatten이 False이면 어절을 보존.

# Komoran (Cont.)

## ■ Komoran 형태소 분석기를 이용해 Tagging하는 예

```
1 text = u"""\u00ac름답지만 다소 복잡하기도한 한국어는 전세계에서 13번째로 많이 사용되는 언어입니다."""
```

```
1 from konlpy.tag import Komoran  
2 komoran = Komoran()
```

```
1 print(komoran.morphs(text))
```

```
['아름답', '지만', '다소', '복잡', '하', '기', '도', '하', 'ㄴ', '한국어', '는', '전', '세계', '에서', '13',  
'번', '째', '로', '많이', '사용', '되', '는', '언어', '이', 'ㅂ니다', '.']
```

```
1 print(komoran.nouns(text))
```

```
['한국어', '전', '세계', '번', '사용', '언어']
```

```
1 print(komoran.pos(text))
```

```
[('아름답', 'VA'), ('지만', 'EC'), ('다소', 'MAG'), ('복잡', 'XR'), ('하', 'XSA'), ('기', 'ETN'), ('도', 'JX'),  
('하', 'VV'), ('ㄴ', 'ETM'), ('한국어', 'NNP'), ('는', 'JX'), ('전', 'NNG'), ('세계', 'NNG'), ('에서', 'JKB'),  
('13', 'SN'), ('번', 'NNB'), ('째', 'XSN'), ('로', 'JKB'), ('많이', 'MAG'), ('사용', 'NNG'), ('되', 'XSV'),  
('는', 'ETM'), ('언어', 'NNG'), ('이', 'VCP'), ('ㅂ니다', 'EF'), ('.', 'SF')]
```

# Okt

## ■ 기존의 Twitter가 0.5.0 부터 이름 변경됨.

```
1 text = u"""아름답지만 다소 복잡하기도한 한국어는 전세계에서 13번째로 많이 사용되는 언어입니다."""
```

```
1 from konlpy.tag import Okt  
2 okt=Okt()  
3 print(okt.morphs(text))
```

```
['아름답지만', '다소', '복잡하', '기도', '한', '한국어', '는', '전세계', '에서', '13', '번', '째', '로', '많이', '사용', '되는',  
'언어', '입니다', '.']
```

```
1 print(okt.pos(text))
```

```
[('아름답지만', 'Adjective'), ('다소', 'Noun'), ('복잡하', 'Adjective'), ('기도', 'Noun'), ('한', 'Josa'), ('한국어', 'Noun'),  
('는', 'Josa'), ('전세계', 'Noun'), ('에서', 'Josa'), ('13', 'Number'), ('번', 'Noun'), ('째', 'Suffix'), ('로', 'Josa'), ('많이',  
'Adverb'), ('사용', 'Noun'), ('되는', 'Verb'), ('언어', 'Noun'), ('입니다', 'Adjective'), ('.', 'Punctuation')]
```

```
1 print(okt.nouns(text))
```

```
['다소', '기도', '한국어', '전세계', '번', '사용', '언어']
```

# Kkma

■ 서울대학교 IDS(Intelligent Data Systems) 연구실에서 자연어 처리를 위한 다양한 모듈 및 자료를 구축하기 위한 과제.

```
1 text = u"""아름답지만 다소 복잡하기도한 한국어는 전세계에서 13번째로 많이 사용되는 언어입니다."""
```

```
1 from konlpy.tag import Kkma  
2 kkma = Kkma()  
3 print(kkma.morphs(text))
```

```
['아름답', '지만', '다소', '복잡', '하', '기', '도', '한', '한국어', '는', '전세계', '에서', '13', '번째', '로', '많이', '사용', '되', '는', '언어', '이', 'ㅂ니다', '.']
```

```
1 print(kkma.pos(text))
```

```
[('아름답', 'VA'), ('지만', 'ECE'), ('다소', 'MAG'), ('복잡', 'NNG'), ('하', 'XSV'), ('기', 'ETN'), ('도', 'JX'), ('한', 'MDN'), ('한국어', 'NNG'), ('는', 'JX'), ('전세계', 'NNG'), ('에서', 'JKM'), ('13', 'NR'), ('번째', 'NNB'), ('로', 'JKM'), ('많이', 'MAG'), ('사용', 'NNG'), ('되', 'XSY'), ('는', 'ETD'), ('언어', 'NNG'), ('이', 'VCP'), ('ㅂ니다', 'EFN'), ('.', 'SF')]
```

```
1 print(kkma.nouns(text))
```

```
['복잡', '한국어', '전세계', '13', '13번째', '번째', '사용', '언어']
```

# Hannanum

- 1999년 KAIST SWRC(Semantic Web Research Center)에 의해 개발된 Java로 만들어진 형태소 분석기  
`konlpy.tag.Hannanum(jvmpath = None)`

메소드	설명
<code>analyze(phrase)</code>	이 분석기는 각 토큰에 대한 다양한 형태 학적후보를 반환합니다.
<code>morphs(phrase)</code>	형태소 분석 문구를 반환합니다.
<code>nouns(phrase)</code>	명사를 추출합니다.
<code>pos(phrase, ntags=9, flatten=True)</code>	<code>ntags</code> 는 태그의 수. 9(한 문자) 또는 22(두 문자)일 수 있습니다. <code>flatten</code> 이 <code>False</code> 이면 어절을 보존합니다.

# 한국어 Tokenizing & POS Tagging Exercise

```
1 text = u"""하지만 한국어는 영어와는 달리 띄어쓰기만으로는 토큰화를 하기에 부족합니다."""
```

```
1 from konlpy.tag import Okt  
2 okt=Okt()  
3 print(okt.morphs(text))
```

```
['하지만', '한국어', '는', '영어', '와는', '달리', '띄어쓰기', '만으로는', '토큰', '화', '를', '하기에', '부족합니다', '.']
```

```
1 print(okt.pos(text))
```

```
[('하지만', 'Conjunction'), ('한국어', 'Noun'), ('는', 'Josa'), ('영어', 'Noun'), ('와는', 'Josa'), ('달리', 'Noun'), ('띄어쓰기', 'Noun'), ('만으로는', 'Josa'), ('토큰', 'Noun'), ('화', 'Suffix'), ('를', 'Josa'), ('하기에', 'Verb'), ('부족합니다', 'Adjective'), ('.', 'Punctuation')]
```

```
1 print(okt.nouns(text))
```

```
['한국어', '영어', '달리', '띄어쓰기', '토큰']
```

# 말뭉치

- 컴퓨터를 이용해 자연어 분석 작업을 할 수 있도록 만든 문서 집합
- KoNLPy 패키지를 설치하고 사용할 수 있는 말뭉치
  - kolaw
    - 한국 법률 말뭉치
    - constitution.txt
  - kobill
    - 대한민국 국회 의안 말뭉치
    - 파일 ID는 의안 번호를 의미
    - 1809890.txt - 1809899.txt

# 말뭉치 (Cont.)

```
1 from konlpy.corpus import kolaw  
2 c = kolaw.open('constitution.txt').read()  
3 print(c[:100])
```

대한민국헌법

유구한 역사와 전통에 빛나는 우리 대한국민은 3·1운동으로 건립된 대한민국임시정부의 법통과 불의에 항거한 4·19민주이념을 계승하고, 조국의 민주개혁과 평화적 통일의

```
1 from konlpy.corpus import kobill  
2 d = kobill.open('1809890.txt').read()  
3 print(d[150:300])
```

초등학교 저학년의 경우에도 부모의 따뜻한 사랑과 보살핌이 필요  
한 나이이나, 현재 공무원이 자녀를 양육하기 위하여 육아휴직을 할  
수 있는 자녀의 나이는 만 6세 이하로 되어 있어 초등학교 저학년인  
자녀를 돌보기 위해서는 해당 부모님은 일자리를 그만 두어야

---

# **Text Preprocessing**

## **Tokenization**



# Tokenization

- 자연어 분석을 위해 문자열을 작은 단위(token)로 나누기
- Token → 나눈 작은 문자열 단위
- 단어 토큰화(Word Tokenization)
- 토큰화시 고려사항
  - 구두점이나 특수 문자를 단순 제외는 안됨.
  - 줄임말과 단어 내에 띄어쓰기가 있는 경우
- 문장 토큰화(Sentence Tokenization)
- 이진 분류기(Binary Classifier)

# Tokenization (Cont.)

- 단어 토큰화(Word Tokenization)
- Token의 기준이 단어(Word)인 경우
- 단어(Word)외에 단어구, 의미를 갖는 문자열도 포함.
- word\_tokenize()

```
1 from nltk.tokenize import word_tokenize  
2 word_tokenize(emma[50:100])
```

```
['Emma',  
'Woodhouse',  
'',  
'handsome',  
'',  
'clever',  
'',  
'and',  
'rich',  
'',  
'with',  
'a']
```

# Tokenization (Cont.)

## ■ word\_tokenize() 사용시

```
1 text = "Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."
```

```
1 import nltk  
2 from nltk.tokenize import word_tokenize  
3 print(word_tokenize(text))
```

```
['Do', "n't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr.', 'Jone', "'s", 'Orphanage', 'is', 'as', 'cheery',  
'as', 'cheery', 'goes', 'for', 'a', 'pastry', 'shop', '.']
```

# Tokenization (Cont.)

## ■ WordPunctTokenizer 사용시

```
1 text = "Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."
```

```
1 import nltk
2 from nltk.tokenize import WordPunctTokenizer
3 WordPunctTokenizer().tokenize(text)
```

```
[ 'Don',
  "'",
  't',
  'be',
  'fooled',
  'by',
  'the',
  'dark',
  'sounding',
  'name',
  "'",
  'Mr',
  "'",
  'Jone',
  "'",
  's',
  'Orphanage',
  'is',
  'as',
  'cheery',
  'as',
  'cheery',
  'goes',
  'for',
  'a',
  'pastry',
  'shop',
  "'"]
```

# Tokenizing (Cont.)

- Token化에 포함할 문자들을 정규표현식을 이용해서 지정하기
- **RegexpTokenizer()**

```
1 from nltk.tokenize import RegexpTokenizer  
2 retokenize = RegexpTokenizer('[\w]+')  
3 retokenize.tokenize(emma[50:100])
```

```
['Emma', 'Woodhouse', 'handsome', 'clever', 'and', 'rich', 'with', 'a']
```

# Tokenizing 시 고려할 사항

- 구두점이나 특수 문자를 단순 제외해서는 안된다.
- 구두점(.)은 문자의 경계를 설명하기 때문에 구두점 제외에 고려해야.
- 단어 자체가 갖고 있는 구두점
  - . Ph.D
  - \$ \$45.44
  - / 01/02/06
  - & AT&T
  - , 123,456,789

# Tokenizing 시 고려할 사항 (Cont.)

- 줄임말과 단어 내에 띄어쓰기가 있는 경우
- 줄임말
  - ' I'm what're we're
- 단어 자체의 띄어쓰기 내제
  - New Work
  - rock 'n' roll

# Tokenizing 시 고려할 사항 (Cont.)

```
1 text = """Starting a home-based restaurant may be an ideal. it doesn't have a food  
2 chain or restaurant of their own."""
```

```
1 import nltk  
2 from nltk.tokenize import TreebankWordTokenizer  
3 tokenizer=TreebankWordTokenizer()  
4 tokenizer.tokenize(text)
```

```
['Starting',  
'a',  
'home-based',  
'restaurant',  
'may',  
'be',  
'an',  
'ideal.',  
'it',  
'does',  
"n't",  
'have',  
'a',  
'food',  
'chain',  
'or',  
'restaurant',  
'of',  
'their',  
'own',  
.']
```

# Sentence Tokenization

- Token의 단위가 문장일 때
- NLTK에서는 **sent\_tokenize()** 지원

```
1 from nltk.tokenize import sent_tokenize  
2 sent_tokenize(emma[:1000])[3]
```

"Sixteen years had Miss Taylor been in Mr. Woodhouse's family, unless as a governess than a friend, very fond of both daughters, but particularly of Emma."

# Sentence Tokenization (Cont.)

- Token의 단위가 문장일 때
- NLTK에서는 **sent\_tokenize()** 지원

```
1 from nltk.tokenize import sent_tokenize  
2 sent_tokenize(emma[:1000])[3]
```

"Sixteen years had Miss Taylor been in Mr. Woodhouse's family, unless as a governess than a friend, very fond of both daughters, but particularly of Emma."

```
1 sentence = """His barber kept his word. But keeping such a huge secret to himself was driving him crazy.  
2 Finally, the barber went up a mountain and almost to the edge of a cliff.  
3 He dug a hole in the midst of some reeds. He looked about, to make sure no one was near."""
```

```
1 import nltk  
2 from nltk.tokenize import sent_tokenize  
3 print(sent_tokenize(sentence))
```

['His barber kept his word.', 'But keeping such a huge secret to himself was driving him crazy.', 'Finally, the barber went up a mountain and almost to the edge of a cliff.', 'He dug a hole in the midst of some reeds.', 'He looked about, to make sure no one was near.']}

# Sentence Tokenization (Cont.)

## ■ 구두점이 여러 개 있을 경우

```
1 import nltk  
2 from nltk.tokenize import sent_tokenize  
3 text="I am actively looking for Ph.D. students. and you are a Ph.D student."  
4 print(sent_tokenize(text))
```

[ 'I am actively looking for Ph.D. students.', 'and you are a Ph.D student.' ]

# 영문 Tokenizing & POS Tagging Exercise

```
1 text="I am actively looking for Ph.D. students. and you are a Ph.D. student."
```

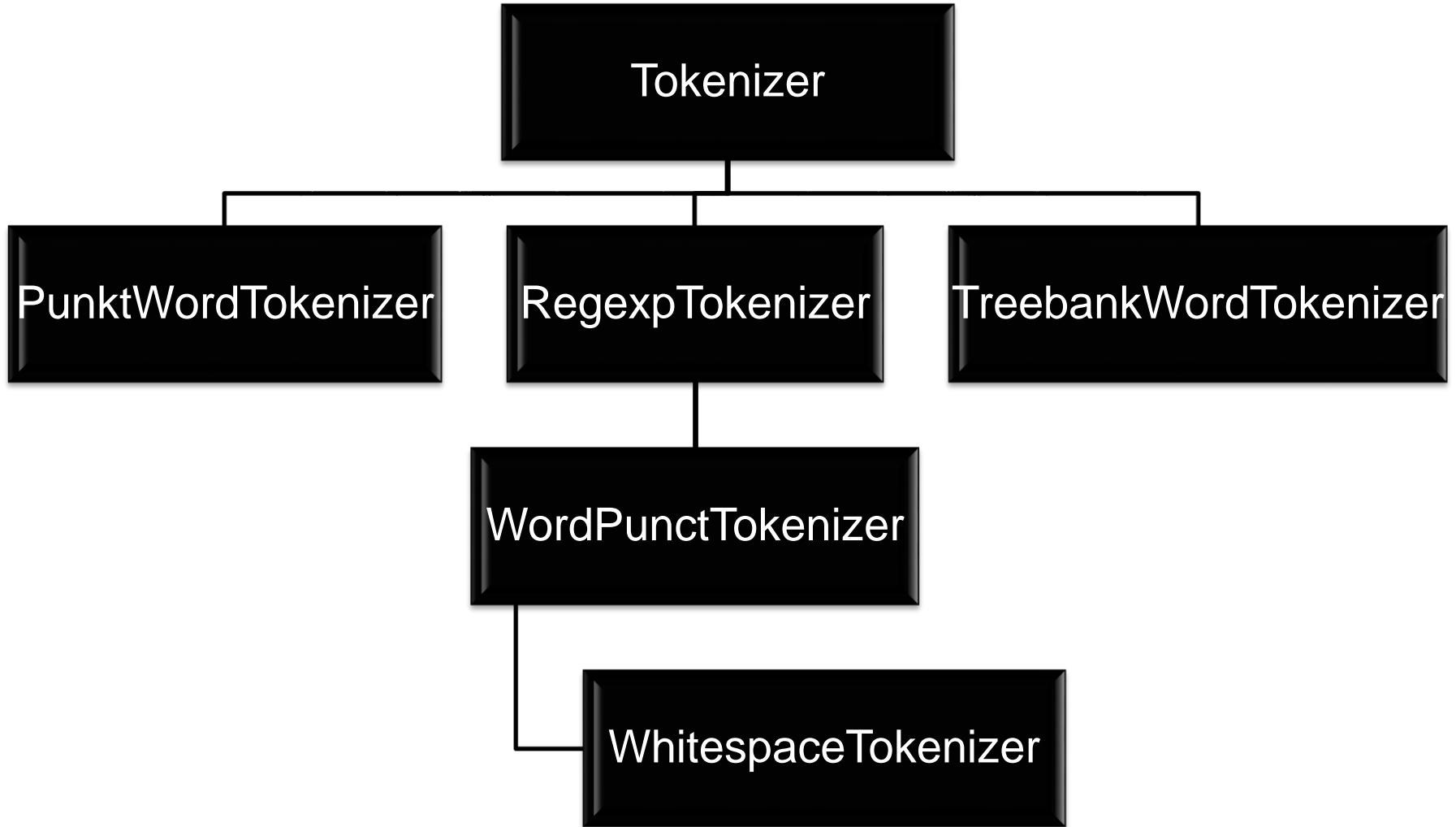
```
1 import nltk  
2 from nltk.tokenize import word_tokenize  
3 print(word_tokenize(text))
```

```
['I', 'am', 'actively', 'looking', 'for', 'Ph.D.', 'students', '.', 'and', 'you', 'are', 'a', 'Ph.D.', 'student', '.']
```

```
1 from nltk.tag import pos_tag  
2 x=word_tokenize(text)  
3 pos_tag(x)
```

```
[('I', 'PRP'),  
 ('am', 'VBP'),  
 ('actively', 'RB'),  
 ('looking', 'VBG'),  
 ('for', 'IN'),  
 ('Ph.D.', 'NNP'),  
 ('students', 'NNS'),  
 ('.', '.'),  
 ('and', 'CC'),  
 ('you', 'PRP'),  
 ('are', 'VBP'),  
 ('a', 'DT'),  
 ('Ph.D.', 'NNP'),  
 ('student', 'NN'),  
 ('.', '.')]
```

# Tokenizer의 상속 Tree



# 이진 분류기(Binary Classifier)

- 문장 토큰화에서의 예외 사항을 발생시키는 구두점의 처리를 위해 입력에 따라 두 개의 Class로 분류하기도.
- Class
  - Period가 단어의 일부분일 경우, 약어(Abbreviation)인 경우
  - Period가 정말로 문장의 구분자(Boundary)인 경우
- 영어권에서 사용하는 약어 사전(Abbreviation Dictionary)
  - <https://public.oed.com/how-to-use-the-oed/abbreviations/>
- Open Source
  - NLTK, OpenNLP, Stanford CoreNLP, splitta, LingPipe
- 참고 자료
  - <https://tech.grammarly.com/blog/posts/How-to-Split-Sentences.html>

# 한국어에서의 Tokenization의 어려움

## ■ 한국어는 교착어이다.

- 영어의 띄어쓰기와 달리 한국어는 조사가 붙어있음.
- 조사 분리 필요

종류	대표적 언어	특징
교착어	한국어, 일본어, 몽골어	어간에 접사가 붙어 단어를 이루고 의미와 문법적 기능이 정해짐
굴절어	라틴어, 독일어, 러시아어	단어의 형태가 변함으로써 문법적 기능이 정해짐
고립어	영어, 중국어	어순에 따라 단어의 문법적 기능이 정해짐

# 한국어에서의 Tokenization의 어려움 (Cont.)

- 한국어는 띄어쓰기가 영어보다 잘 지켜지지 않는다.
  - 원래 한국어는 띄어쓰기가 없었음.
  - 1933년 한글맞춤법통일안 이후
- 평서문과 의문문
  - 영어와 달리 한국어는 의문문과 평서문이 같은 형태의 문장 구조를 가진다.
  - 물음표나 마침표가 붙지 않으면 잘 알기 어렵다.

언어	평서문	의문문
영어	I ate my lunch.	Did you have lunch?
한국어	점심 먹었어.	점심 먹었어?

# 한국어에서의 Tokenization의 어려움 (Cont.)

## ■ 주어 생략

- 영어와 달리 주어가 생략되는 경우가 많음.
- 번역시 어려움.

## ■ 한자 기반 언어

- 한자의 영향으로 한자의 조합으로 이루어지는 단어들이 많음.
- 단어가 조합되어야 하나의 의미로 인식.

언어	단어	조합
영어	concentrate	con(=together) + centr(=center) + ate(=make)
한국어	집중(集中)	集(모을 집) + 中(가운데 중)

# 형태소 분석(Morphological Analysis)

## ■ 형태소

- 언어학에서 일정한 의미가 있는 가장 작은 말 단위.

## ■ 행태소 분석

- 단어로부터 어근, 접두사, 접미사, 품사 등 다양한 언어적 속성을 파악하고 이를 이용하여 형태소를 찾아내거나 처리하는 작업.

## ■ 어간 추출(Stemming)

## ■ 표제어 추출(Lemmatizing)

## ■ 품사 부착(Part-Of-Speech Tagging)



---

# **Text Preprocessing**

## **Cleaning and Normalization**



# 정제(Cleaning)과 정규화(Normalization)

- Token화 작업 전, 후에 Data를 용도에 맞게 정제 및 정규화 필요
- 정제
  - 갖고 있는 Corpus로부터 Noise Data 제거
- 정규화
  - 표현 방법이 다른 단어들을 통합시켜서 같은 단어로 만듦.
- 규칙에 기반한 표기가 다른 단어들의 통합
  - USA와 US
- 대소문자 통합
- 불필요한 단어 제거

# 불필요한 단어의 제거

## ■ Noise Data

- 정제 작업시 자연어가 아니면서 아무 의미도 갖지 않는 글자들(예, 특수문자)
- 분석하고자 하는 목적에 맞지 않는 불필요한 단어들.

## ■ 방법

- 불용어(Stopword) 제거
- 등장 빈도가 적은 단어 제거
- 길이가 짧은 단어 제거

```
1 # 길이가 1~2인 단어들을 정규 표현식을 이용하여 삭제
2 import re
3 text = "I was wondering if anyone out there could enlighten me on this car."
4 shortword = re.compile(r'[^W]*[^b]w{1,2}[^b]')
5 print(shortword.sub(' ', text))
```

was wondering anyone out there could enlighten this car.



---

# **Text Preprocessing**

## **Stemming & Lemmatization**



# Stemming

- 변화된 단어에서 접사(affix)를 제거하여 같은 의미를 가지는 형태소의 기본형을 찾는 방법.
- NLTK에서 제공하는 Class
  - **PorterStemmer**
  - **LancasterStemmer**
  - **RegexpStemmer**
  - **SnowballStemmer**
- 어간 추출법(Stemming)은 단순히 어미를 제거할 뿐이므로 단어의 원형을 정확히 찾아주지는 않는다.

# PorterStemmer Class

- Martin Porter의 Porter Stemming Algorithm 구현
- 영어의 접미사(suffix) 제거

```
words = ['sending', 'cooking', 'flies', 'lives', 'crying', 'dying']
```

```
from nltk.stem import PorterStemmer  
ps = PorterStemmer()  
[ps.stem(word) for word in words]
```

```
['send', 'cook', 'fli', 'live', 'cri', 'die']
```

# LancasterStemmer Class

- Lancaster 대학이 개발한 Algorithm을 구현.

```
words = ['sending', 'cooking', 'flies', 'lives', 'crying', 'dying']
```

```
from nltk.stem import LancasterStemmer  
ls = LancasterStemmer()  
[ls.stem(word) for word in words]
```

```
['send', 'cook', 'fli', 'liv', 'cry', 'dying']
```

# RegexpStemmer Class

- 정규 표현식이용해서 어간 추출

```
words = ['sending', 'cooking', 'flies', 'lives', 'crying', 'dying']
```

```
from nltk.stem.regexp import RegexpStemmer  
rs = RegexpStemmer(r'ing')  
[rs.stem(word) for word in words]
```

```
['send', 'cook', 'flies', 'lives', 'cry', 'dy']
```

# SnowballStemmer Class

- 영어 포함 15개국 언어에 대한 어간 추출을 지원.

```
words = ['enviar', 'cocina', 'moscas', 'vidas', 'llorar', 'morir']
```

```
from nltk.stem import SnowballStemmer  
ss = SnowballStemmer('spanish')  
[ss.stem(word) for word in words]
```

```
['envi', 'cocin', 'mosc', 'vid', 'llor', 'mor']
```

# 표제어 추출(Lemmatizing)

- 표제어 추출은 어간 추출(Stemming)과 비슷하지만 동의어 대체와 더 유사하다.
- 어간 추출과 달리 원형 복원 후에도 그 단어는 같은 의미를 가진다.
- 품사 지정할 때 더 정확히 원형을 찾을 수 있다.
- NLTK에서는 **WordNetLammatizer** 제공

# WordNetLammatizer

```
1 words=['policy', 'doing', 'organization', 'have', 'going', 'love', 'lives', 'fly', 'dies', 'watched', 'has', 'starting']
```

```
1 import nltk
2 from nltk.stem import WordNetLemmatizer
3 wnl = WordNetLemmatizer()
4 [wnl.lemmatize(word) for word in words]
```

```
['policy',
'doing',
'organization',
'have',
'going',
'love',
'life',
'fly',
'dy',
'watched',
'ha',
'starting']
```

# 한국어 Stemming

- 한국어는 5언 9품사의 구조임.

언	품사
체언	명사, 대명사, 수사
수식언	관형사, 부사
관계언	조사
독립언	감탄사
용언	동사, 형용사

---

# **Text Preprocessing**

## **Stopword**



# 불용어(Stopword)

- 큰 의미가 없는 단어
- 문장 내에서 자주 등장하지만 문장 분석에 큰 도움이 되지 않는 단어
- I, my, me, over, 조사, 접미사 등.
- NLTK에서는 100여개 이상의 영어 단어들 등록

```
1 import nltk  
2 from nltk.corpus import stopwords  
3 stopwords.words('english')[:20]
```

```
['i',  
'me',  
'my',  
'myself',  
'we',  
'our',  
'ours',  
'ourselves',  
'you',  
"you're",  
"you've",  
"you'll",  
"you'd",  
'your',  
'yours',  
'yourself',  
'yourselves',  
'he',  
'him',  
'his']
```

# 불용어(Stopword) (Cont.)

```
1 import nltk
2 from nltk.corpus import stopwords
3 from nltk.tokenize import word_tokenize
4
5 example = "Family is not an important thing. It's everything."
6 stop_words = set(stopwords.words('english'))
7
8 word_tokens = word_tokenize(example)
9
10 result = []
11 for w in word_tokens:
12     if w not in stop_words:
13         result.append(w)
14
15 print(word_tokens)
16 print(result)
```

```
['Family', 'is', 'not', 'an', 'important', 'thing', '.', 'It', "'s", 'everything', '.']
['Family', 'important', 'thing', '.', 'It', "'s", 'everything', '.']
```

# 불용어(Stopword)

- 큰 의미가 없는 단어
- 문장 내에서 자주 등장하지만 문장 분석에 큰 도움이 되지 않는 단어
- 보통 형태소 분석 후, 조사 / 접속사 등을 제거하는 방법
- 현업에서 사용자가 직접 불용어 사전을 만들어 사용
- 보편적으로 선택할 수 있는 한국어 불용어 리스트
  - <https://www.ranks.nl/stopwords/korean>
- 한국어용 불용어를 제거하는 가장 좋은 방법은 코드 내에서 직접 정의하지 않고 txt 파일이나 csv 파일로 수많은 불용어를 정리해놓고, 이를 불러와서 사용하는 방법 추천.

# 불용어(Stopword) (Cont.)

```
1 import nltk
2 from nltk.corpus import stopwords
3 from nltk.tokenize import word_tokenize
4
5 example = "고기를 아무렇게나 구우려고 하면 안 돼. 고기라고 다 같은 게 아니거든. 예컨대 삼겹살을 구울 때는 중요한 게 있지."
6 stop_words = "아무거나 아무렇게나 어찌하든지 같다 비슷하다 예컨대 이럴정도로 하면 아니거든"
7
8 stop_words=stop_words.split(' ')
9
10 word_tokens = word_tokenize(example)
11
12 result = []
13 for w in word_tokens:
14     if w not in stop_words:
15         result.append(w)
16 # 위의 4줄 코드는 아래의 한 줄로 대체 가능
17 # result=[word for word in word_tokens if not word in stop_words]
18
19 print(word_tokens)
20 print(result)
```

```
['고기를', '아무렇게나', '구우려고', '하면', '안', '돼', '.', '고기라고', '다', '같은', '게', '아니거든', '.', '예컨대', '삼겹살', '을', '구울', '때는', '중요한', '게', '있지', '.']
```

```
['고기를', '구우려고', '안', '돼', '.', '고기라고', '다', '같은', '게', '.', '삼겹살을', '구울', '때는', '중요한', '게', '있지', '.']
```

# **Text Preprocessing**

## **Regular Expression**



---

# **Text Preprocessing**

## **Splitting Data**



# Splitting Data

- ML or DL의 Model에서 Data를 사용하기 위해서는 Data를 적절히 분리해 놓는 작업이 필요.
- Spam 분류기

Text>Email의 내용	Label(Spam 여부)
당신에게 드리는 마지막 혜택...	Yes
내일 볼 수 있을지 확인 부탁...	No
...	...
(광고) 멋있어질 수 있는 ...	Yes

# Splitting Data (Cont.)

- 전제 : 문제지 20,000개
  - 훈련 데이터
    - X\_train : 문제지 데이터(18,000개)
    - y\_train : 문제지에 대한 정답 데이터(2,000개)
  - 테스트 데이터
    - X\_test : 시험지 데이터
    - y\_test : 시험지에 대한 정답 데이터
- 기계는 18,000개의 문제를 2000개의 답을 보면서 학습한다.
- 학습이 끝난 기계에게 y\_test를 보여주지 않고 X\_test에 대해서 정답을 예측하게 한다.
- 기계가 예측한 답과 실제 정답인 y\_test를 비교하면서 정확도(Accuracy)를 평가한다.

# X(X\_train)과 y(y\_train) 분리하기

## zip() 함수를 이용하여 분리하기

### ■ zip()

- 동일한 개수를 가지는 Sequence 자료형에서 각 순서에 등장하는 원소들끼리 묶어주는 역할 수행.
- List의 리스트 구성에서 zip 함수는 x와 y를 분리하는데 유용.

```
1 X, y = zip(['a', 1], ['b', 2], ['c', 3])
2 print(X)
3 print(y)
```

```
('a', 'b', 'c')
(1, 2, 3)
```

# X(X\_train)과 y(y\_train) 분리하기 (Cont.)

## zip() 함수를 이용하여 분리하기

### ■ zip()

```
| 1 sequences=[['a', 1], ['b', 2], ['c', 3]] # 리스트의 리스트 또는 행렬 또는 뒤에서 배울 개념인 2D Tensor.  
| 2 X,y = zip(*sequences) # *를 추가  
| 3 print(X)  
| 4 print(y)
```

```
('a', 'b', 'c')  
(1, 2, 3)
```

- 각 데이터에서 첫 번째로 등장한 원소들끼리 묶이고, 두 번째로 등장한 원소들끼리 묶인 것을 볼 수 있다.
- 이를 각각 X데이터와 y데이터로 사용할 수 있다.

# X(X\_train)과 y(y\_train) 분리하기 (Cont.)

## DataFrame을 이용하여 분리하기

- **DataFrame**은 열의 이름으로 각 열에 접근이 가능하므로, 이를 이용하면 손쉽게 X 데이터와 y 데이터를 분리할 수 있다.

```
1 import pandas as pd
2
3 values = [['당신에게 드리는 마지막 혜택!', 1],
4            ['내일 봄 수 있을지 확인 부탁드...', 0],
5            ['도연씨. 잘 지내시죠? 오랜만입...', 0],
6            ['(광고) AI로 주가를 예측할 수 있다!', 1]]
7 columns = ['메일 본문', '스팸 메일 유무']
8
9 df = pd.DataFrame(values, columns=columns)
10 df
```

메일 본문 스팸 메일 유무

	메일 본문	스팸 메일 유무
0	당신에게 드리는 마지막 혜택!	1
1	내일 봄 수 있을지 확인 부탁드...	0
2	도연씨. 잘 지내시죠? 오랜만입...	0
3	(광고) AI로 주가를 예측할 수 있다!	1

# X(X\_train)과 y(y\_train) 분리하기 (Cont.)

## DataFrame을 이용하여 분리하기

### ■ X 데이터 출력

```
1 X=df['메일 본문']
2 y=df['스팸 메일 유무']
3 X
```

```
0      당신에게 드리는 마지막 혜택!
1      내일 될 수 있을지 확인 부탁드...
2      도연씨. 잘 지내시죠? 오랜만입...
3      (광고) AI로 주가를 예측할 수 있다!
Name: 메일 본문, dtype: object
```

# X(X\_train)과 y(y\_train) 분리하기 (Cont.)

## DataFrame을 이용하여 분리하기

- y 데이터 출력

1	y
0	1
1	0
2	0
3	1

Name: 스팸 메일 유무, dtype: int64

# X(X\_train)과 y(y\_train) 분리하기 (Cont.)

## Numpy를 이용하여 분리하기

```
1 import numpy as np  
2 ar = np.arange(0,16).reshape((4,4))  
3 print(ar)
```

```
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]  
 [12 13 14 15]]
```

```
1 X=ar[:, :3]  
2 print(X)
```

```
[[ 0  1  2]  
 [ 4  5  6]  
 [ 8  9 10]  
 [12 13 14]]
```

```
1 y=ar[:,3]  
2 print(y)
```

```
[ 3  7 11 15]
```

# Test Data 분리하기

## Scikit-learn을 이용하여 분리하기

- Scikit-learn은 학습용 테스트와 테스트용 데이터를 분리하게 해주는 **train\_test\_split()** 함수를 지원.

```
1 from sklearn.model_selection import train_test_split  
2 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
```

- **x** : 독립 변수 데이터. (배열이나 DataFrame)
- **y** : 종속 변수 데이터. Label Data.
- **test\_size** : 테스트용 데이터 개수를 지정한다. 1보다 작은 실수를 기재할 경우, 비율을 나타낸다.
- **train\_size** : 학습용 데이터의 개수를 지정한다. 1보다 작은 실수를 기재할 경우, 비율을 나타낸다. (**test\_size**와 **train\_size** 중 하나만 기재해도 가능)
- **random\_state** : 난수 seed

# Test Data 분리하기 (Cont.)

## Scikit-learn을 이용하여 분리하기

```
1 import numpy as np
2 from sklearn.model_selection import train_test_split
3 X, y = np.arange(10).reshape((5, 2)), range(5)
4 # 실습을 위해 임의로 X와 y가 이미 분리된 데이터를 생성
5 print(X)
6 print(list(y)) #레이블 데이터
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
[0, 1, 2, 3, 4]
```

```
1 print(X_train)
2 print(X_test)
```

```
[[2 3]
 [4 5]
 [6 7]]
[[8 9]
 [0 1]]
```

```
1 print(y_train)
2 print(y_test)
```

```
[1, 2, 3]
[4, 0]
```

```
1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1234)
2 #3분의 1만 test 데이터로 지정.
3 #random_state 지정으로 인해 순서가 섞인 채로 훈련 데이터와 테스트 데이터가 나눠진다.
```

# Test Data 분리하기 (Cont.)

## 수동으로 분리하기

```
1 import numpy as np  
2 X, y = np.arange(0,24).reshape((12,2)), range(12)  
3 # 실습을 위해 임의로 X와 y가 이미 분리 된 데이터를 생성  
4 print(X)
```

```
[[ 0  1]  
 [ 2  3]  
 [ 4  5]  
 [ 6  7]  
 [ 8  9]  
[10 11]  
[12 13]  
[14 15]  
[16 17]  
[18 19]  
[20 21]  
[22 23]]
```

```
1 print(list(y))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

# Test Data 분리하기 (Cont.)

## 수동으로 분리하기

- 이제 훈련 데이터의 개수와 테스트 데이터의 개수를 정한다.
- **n\_of\_train**은 훈련 데이터의 개수를 의미하며,  
**n\_of\_test**는 테스트 데이터의 개수를 의미한다.

```
1 n_of_train = int(len(X) * 0.8) # 데이터의 전체 길이의 80%에 해당하는 길이값을 구한다.  
2 n_of_test = int(len(X) - n_of_train) # 전체 길이에서 80%에 해당하는 길이를 뺀다.  
3 print(n_of_train)  
4 print(n_of_test)
```

# Test Data 분리하기 (Cont.)

## 수동으로 분리하기

- 주의할 점은 아직 훈련 데이터와 테스트 데이터를 나눈 것이 아니라, 이 두 개의 개수를 몇 개로 할지 정하기만 한 상태이다.
- 또한 여기서 n\_of\_train을 `len(x) * 0.8`로 구했는데, 소수점 이하가 누락될 수 있다.
- 전체 데이터의 개수가 4,518이라고 가정했을 때
  - 4,518의 80% = 3,614.4(정수만 취함 3,614)
  - 4,518의 20% = 903.6(정수만 취함 903)
  - 그러면  $3,614 + 903 = 4517$
  - 데이터 1개 누락

# Test Data 분리하기 (Cont.)

## 수동으로 분리하기

- 실제로 데이터를 나눌 때는 `n_of_train`과 같이 하나의 변수만 사용하면 데이터의 누락을 방지할 수 있다.

```
1 X_test = X[n_of_train:] #전체 데이터 중에서 20%만큼 뒤의 데이터 저장  
2 y_test = y[n_of_train:] #전체 데이터 중에서 20%만큼 뒤의 데이터 저장  
3 X_train = X[:n_of_train] #전체 데이터 중에서 80%만큼 앞의 데이터 저장  
4 y_train = y[:n_of_train] #전체 데이터 중에서 80%만큼 앞의 데이터 저장
```

```
1 print(X_test)  
2 print(list(y_test))
```

```
[[18 19]  
[20 21]  
[22 23]]  
[9, 10, 11]
```

각각 길이가 3이 됨.

# **Text Preprocessing**

## **Integer Encoding**



# 정수(Integer) Encoding

- 자연어 처리에서 Text를 숫자로 바꾸는 여러 가지 기법들 중 하나.
- 또는 그런 기법들을 본격적으로 적용시키기 위한 첫 단계로 각 단어를 고유한 숫자에 Mapping시키는 전처리 작업.
- 갖고 있는 Text에 단어가 5,000개가 있다면, 5,000개의 단어들 각각에 0번부터 4,999번까지 단어와 Mapping되는 고유한 숫자 즉 Index를 부여하는 방법.
- Index를 부여하는 방법
  - Random으로
  - 보통은 전처리도 같이 겸하기 위해 단어에 대한 빈도수로 정렬한 뒤 부여.

# 정수(Integer) Encoding (Cont.)

## ■ 빈도수 순서대로 단어를 정수로 변환

### Step1. 문장 Tokening

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

```
1 # 문장 토큰화
2 from nltk.tokenize import sent_tokenize
3 text = sent_tokenize(text)
4 print(text)
```

```
['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 \n    his word.', 'His barber kept his secret.', 'But keeping and keeping such a huge secret to himself \n    was driving the barber crazy.', 'the barber went up a huge mountain.']}
```

# 정수(Integer) Encoding (Cont.)

## Step2. 정제(Cleaning), 단어 Tokenizing, 각 단어별 빈도수 계산

```
1 from nltk.tokenize import word_tokenize
2 from nltk.corpus import stopwords
3 from collections import Counter
4 vocab = Counter() # Python의 Counter module을 이용하면 단어의 모든 빈도를 쉽게 계산할 수 있다.
5
6 sentences = []
7 stop_words = set(stopwords.words('english'))
8
9 for i in text:
10     sentence = word_tokenize(i) # 단어 토큰화를 수행.
11     result = []
12
13     for word in sentence:
14         word = word.lower() # 모든 단어를 소문자화하여 단어의 개수를 줄인다.
15         if word not in stop_words: # 단어 토큰화 된 결과에 대해서 불용어 제거.
16             if len(word) > 2: # 단어 길이가 2이하인 경우에 대하여 추가로 단어 제거.
17                 result.append(word)
18                 vocab[word] = vocab[word]+1 #각 단어의 빈도를 Count.
19     sentences.append(result)
20 print(sentences)
```

```
[['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']]
```

```
1 print(vocab)
```

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

# 정수(Integer) Encoding (Cont.)

## Step3. 각 단어들을 빈도순으로 정렬

```
1 vocab_sorted=sorted(vocab.items(), key=lambda x:x[1], reverse=True)
2 print(vocab_sorted)
```

```
[('barber', 8), ('secret', 6), ('huge', 5), ('kept', 4), ('person', 3), ('word', 2), ('keeping', 2), ('good', 1), ('knew', 1), ('driving', 1), ('crazy', 1), ('went', 1), ('mountain', 1)]
```

## Step4. 높은 빈도수에 따라 Index 부여

```
1 word_to_index={}
2 i=0
3 for (word, frequency) in vocab_sorted :
4     if frequency > 1 : # 정제(Cleaning) 챕터에서 언급했듯이 빈도수가 적은 단어는 제외한다.
5         i=i+1
6         word_to_index[word]=i
7 print(word_to_index)
```

```
{'barber': 1, 'secret': 2, 'huge': 3, 'kept': 4, 'person': 5, 'word': 6, 'keeping': 7}
```

- 1의 Index가 가장 빈도수가 높다.

# Keras의 Text 전처리

## ■ fit\_on\_texts()

- Text의 list를 가지고 단어 빈도수에 기반한 사전 생성

```
1 from keras.preprocessing.text import Tokenizer
2 text=["A barber is a person. a barber is good person. a barber is huge person. #
3     he Knew A Secret! The Secret He Kept is huge secret. Huge secret. His barber kept his word. #
4     a barber kept his word. His barber kept his secret. But keeping and keeping such a huge secret #
5     to himself was driving the barber crazy. the barber went up a huge mountain."]
6 t = Tokenizer()
7 t.fit_on_texts(text)
8 print(t.word_index)

{'a': 1, 'barber': 2, 'secret': 3, 'huge': 4, 'his': 5, 'is': 6, 'kept': 7, 'person': 8, 'the': 9, 'he': 10, 'word': 11, 'keeping':
12, 'good': 13, 'knew': 14, 'but': 15, 'and': 16, 'such': 17, 'to': 18, 'himself': 19, 'was': 20, 'driving': 21, 'crazy': 22, 'wen
t': 23, 'up': 24, 'mountain': 25}
```

## ■ word\_index

- 각 단어에 부여된 Index 확인

```
1 print(t.word_index)

{'a': 1, 'barber': 2, 'secret': 3, 'huge': 4, 'his': 5, 'is': 6, 'kept': 7, 'person': 8, 'the': 9, 'he': 10, 'word': 11, 'keepin
g': 12, 'good': 13, 'knew': 14, 'but': 15, 'and': 16, 'such': 17, 'to': 18, 'himself': 19, 'was': 20, 'driving': 21, 'crazy': 22,
'went': 23, 'up': 24, 'mountain': 25}
```

# Keras의 Text 전처리 (Cont.)

## ■ word\_counts

- 단어별 개수 파악

```
1 print(t.word_counts)
```

```
OrderedDict([('a', 8), ('barber', 8), ('is', 4), ('person', 3), ('good', 1), ('huge', 5), ('he', 2), ('knew', 1), ('secret', 6), ('the', 3), ('kept', 4), ('his', 5), ('word', 2), ('but', 1), ('keeping', 2), ('and', 1), ('such', 1), ('to', 1), ('himself', 1), ('was', 1), ('driving', 1), ('crazy', 1), ('went', 1), ('up', 1), ('mountain', 1)])
```

## ■ texts\_to\_sequences()

- 입력으로 들어온 Corpus를 word\_index에서 정해진 Index에 맞춰서 변환하여 출력

```
1 print(t.texts_to_sequences(text))
```

```
[[1, 2, 6, 1, 8, 1, 2, 6, 13, 8, 1, 2, 6, 4, 8, 10, 14, 1, 3, 9, 3, 10, 7, 6, 4, 3, 4, 3, 5, 2, 7, 5, 11, 1, 2, 7, 5, 11, 5, 2, 7, 5, 3, 15, 12, 16, 12, 17, 1, 4, 3, 18, 19, 20, 21, 9, 2, 22, 9, 2, 23, 24, 1, 4, 25]]
```

# Keras의 Text 전처리 (Cont.)

## ■ 빈도수가 1인 단어 제거

```
1 words_frequency = [w for w,c in t.word_counts.items() if c < 2] # 빈도수가 2이하인 단어를 w라고 저장
2 for w in words_frequency:
3     del t.word_index[w] # 해당 단어에 대한 인덱스 정보를 삭제
4     del t.word_counts[w] # 해당 단어에 대한 카운트 정보를 삭제
5 print(t.texts_to_sequences(text))
6 print(t.word_index)
```

```
[[1, 2, 6, 1, 8, 1, 2, 6, 8, 1, 2, 6, 4, 8, 10, 1, 3, 9, 3, 10, 7, 6, 4, 3, 4, 3, 5, 2, 7, 5, 11, 1, 2, 7, 5, 11, 5, 2, 7, 5, 3, 1
2, 12, 1, 4, 3, 9, 2, 9, 2, 1, 4]]
{'a': 1, 'barber': 2, 'secret': 3, 'huge': 4, 'his': 5, 'is': 6, 'kept': 7, 'person': 8, 'the': 9, 'he': 10, 'word': 11, 'keeping': 12}
```

# enumerate()

- 주어진 단어의 list에 대해서 쉽게 Index를 부여할 수 있다.
- 주어진 입력에 Index를 부여하여 Index도 함께 반환 해준다는 특징이 있다.

```
1 test=[8, 2, 5, 1, 3, 7, 9, 4, 6, 10]
2
3 for index, value in enumerate(test): # 입력의 순서대로 0부터 인덱스를 부여함.
4     print("index : {}, value: {}".format(index,value))
```

```
index : 0, value: 8
index : 1, value: 2
index : 2, value: 5
index : 3, value: 1
index : 4, value: 3
index : 5, value: 7
index : 6, value: 9
index : 7, value: 4
index : 8, value: 6
index : 9, value: 10
```

## enumerate() (Cont.)

- 단어를 우선 정렬 하고, enumerate()를 사용하면 Integer Encoding 결과를 쉽게 얻을 수 있다.
- 다음 코드는 주어진 데이터로부터 단어를 빈도수로 정렬하고, enumerate()로 정수 Encoding을 하는 과정까지 진행한다.

```
1 # 문장 토큰화까지는 되어 있다고 가정함
2 text=[['barber', 'person'], ['barber', 'good', 'person'], ['barber', 'huge', 'person'], ['knew', 'secret'], #
3     ['secret', 'kept', 'huge', 'secret'], ['huge', 'secret'], ['barber', 'kept', 'word'], #
4     ['barber', 'kept', 'word'], ['barber', 'kept', 'secret'], #
5     ['keeping', 'keeping', 'huge', 'secret', 'driving', 'barber', 'crazy'], ['barber', 'went', 'huge', 'mountain']]
```

- 단어 집합(vocabulary)을 만들기 위해서 문장의 경계인 [, ]를 제거하고 단어들을 하나의 list로 만든다.

```
1 vocab = sum(text, [])
2 print(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', 'keeping', 'huge', 'secret',
'driving', 'barber', 'crazy', 'barber', 'went', 'huge', 'mountain']
```

# enumerate() (Cont.)

- 단어를 빈도순으로 정렬하고, 중복도 제거한 결과를 만든다.

```
1 from nltk.tokenize import word_tokenize
2 from nltk.corpus import stopwords
3 from collections import Counter
4 myvocab = Counter() # Python의 Counter 모듈을 이용하면 단어의 모든 빈도를 쉽게 계산할 수 있습니다.
5 stop_words = set(stopwords.words('english'))
6 for word in vocab:
7     word = word.lower() # 모든 단어를 소문자화하여 단어의 개수를 줄입니다.
8     if word not in stop_words: # 단어 토큰화 된 결과에 대해서 불용어를 제거합니다.
9         myvocab[word]=myvocab[word]+1 #각 단어의 빈도를 Count 합니다.
10 print(myvocab)
```

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

- enumerate()를 통해 순서대로 Index를 부여한다.

```
1 word_to_index = {word : index+1 for index, word in enumerate(myvocab)}
2 # 인덱스를 0이 아닌 1부터 부여.
3 print(word_to_index)

{'barber': 1, 'person': 2, 'good': 3, 'huge': 4, 'knew': 5, 'secret': 6, 'kept': 7, 'word': 8, 'keeping': 9, 'driving': 10, 'crazy': 11, 'went': 12, 'mountain': 13}
```

# NLTK의 FreqDist Class

- NLTK에서는 Token들의 빈도를 손쉽게 셀 수 있도록 지원하는 빈도수 계산 클래스
- Token들을 입력으로 받아 해당 Token의 빈도 정보를 계산.

```
1 test_list = ['barber', 'barber', 'person', 'barber', 'good', 'person']
2 from nltk import FreqDist
3 fdist = FreqDist(test_list)
```

```
1 fdist.N() # 전체 단어 개수 출력
```

6

```
1 fdist.freq("barber") # 'barber'라는 단어의 확률.
```

0.5

```
1 fdist["barber"] # 'barber'라는 단어의 빈도수 출력
```

3

```
1 fdist.most_common(2) # 등장 빈도수가 높은 상위 2개의 단어만 출력
```

[('barber', 3), ('person', 2)]

---

# **Text Preprocessing**

## **One-Hot Encoding**



# One-Hot Encoding

- 자연어 처리에서 문자를 숫자로 바꾸는 여러 가지 기법들중 하나.
- 가장 기본적인 표현 방법이며, Machine Learning, Deep Learning을 하기 위해서는 반드시 배워야 하는 표현 방법.
- Vocabulary(단어집합)
  - 서로 다른 단어들의 집합
  - book과 books는 서로 다른 단어로 간주.

# One-Hot Encoding (Cont.)

- 먼저 해야 할 일은 갖고 있는 우선 단어 집합을 만드는 일
- Text의 모든 단어를 중복을 허락하지 않고 모아놓는다면 이를 단어 집합이라고 한다.
- 그리고 이 단어 집합에 고유한 숫자를 부여하는 일을 진행한다.
- 갖고 있는 텍스트에 단어가 5,000개면 단어 집합의 크기도 5,000이라고 한다.
- 5,000개의 단어가 있는 이 단어 집합의 단어들마다 0번부터 4,999번까지 Index를 부여한다.

# One-Hot Encoding (Cont.)

- 단어 집합의 크기를 벡터의 차원으로 하고, 표현하고 싶은 단어의 Index에 1의 값을 부여하고, 다른 Index에는 0을 부여하는 단어의 벡터 표현.
- 이렇게 해서 One-Hot Vector를 만든다.
- Encoding 과정.
  1. 각 단어에 고유한 Index를 부여(정수 Encoding)
  2. 표현하고 싶은 단어의 Index의 위치에 1을 부여하고, 다른 단어의 Index의 위치에는 0을 부여.

# One-Hot Encoding (Cont.)

```
1 from konlpy.tag import Okt  
2 okt = Okt()  
3 token = okt.morphs("나는 자연어 처리를 배운다")  
4 print(token)
```

['나', '는', '자연어', '처리', '를', '배운다']

```
1 word2index={}  
2 for voca in token:  
3     if voca not in word2index.keys():  
4         word2index[voca]=len(word2index)  
5 print(word2index)
```

{'나': 0, '는': 1, '자연어': 2, '처리': 3, '를': 4, '배운다': 5}

```
1 def one_hot_encoding(word, word2index):  
2     one_hot_vector = [0]*(len(word2index))  
3     index = word2index[word]  
4     one_hot_vector[index] = 1  
5     return one_hot_vector  
6  
7 one_hot_encoding("자연어",word2index)
```

[0, 0, 1, 0, 0, 0]

# Keras를 이용한 One-Hot Encoding

## ■ to\_categorical()

- Keras에서 자동으로 One-Hot Encoding을 만들어 주는 유용한 도구

## ■ 정수 Encoding

```
1 text="나랑 점심 먹으려 갈래 점심 메뉴는 햄버거 갈래 갈래 햄버거 최고야"
2 from keras.preprocessing.text import Tokenizer
3 t = Tokenizer()
4 t.fit_on_texts([text])
5
6 print(t.word_index)
```

```
{'갈래': 1, '점심': 2, '햄버거': 3, '나랑': 4, '먹으려': 5, '메뉴는': 6, '최고야': 7}
```

```
1 text2 = "점심 먹으려 갈래 메뉴는 햄버거 최고야"
2 x=t.texts_to_sequences([text2])
3 print(x)
```

```
[[2, 5, 1, 6, 3, 7]]
```

# Keras를 이용한 One-Hot Encoding (Cont.)

```
1 vocab_size = len(t.word_index) # 단어 집합의 크기. 이 경우는 단어의 개수가 70으로 7.  
2  
3 from keras.utils import to_categorical  
4 x = to_categorical(x, num_classes=vocab_size+1) # 실제 단어 집합의 크기보다 +1로 크기를 만들어야함.  
5  
6 print(x)
```

```
[[[0., 0., 1., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 1., 0., 0.],  
 [0., 1., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 1., 0.],  
 [0., 0., 0., 1., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 1.]]]
```

# Keras를 이용한 One-Hot Encoding (Cont.)

## ■ 주의할 점

- `to_categorical()`은 정수 encoding으로 부여된 Index를 그대로 배열의 Index로 사용
- `t.fit_on_texts()`를 사용하여 정수 encoding시 실제 단어 집합의 크기보다 +1의 크기를 인자로 주어야 한다.
- `t.fit_on_texts()`는 Index를 1부터 부여하기 때문에 배열의 Index가 0부터 시작하므로 맨 마지막 Index를 가진 단어의 Index가 7 이므로, 이를 One-Hot Vector로 만들기 위해서 8의 크기를 가진 배열이 필요.

# One-Hot Encoding (Cont.)

## ■ One-Hot Encoding의 한계

- 단어의 개수가 늘어날 수록, Vector를 저장하기 위해 필요한 공간이 계속 늘어난다.
- One-Hot Vector는 단어 집합의 크기가 곧 Vector의 차원 수가 된다.
- 매우 비효율적 방법
- 단어의 유사성을 전혀 표현하지 못한다는 단점도 있다.
- 검색 시스템 등에서 심각한 문제 유발

# One-Hot Encoding (Cont.)

## ■ One-Hot Encoding의 한계 해결방안

- 단어의 '의미'를 다차원 공간에 Vector화 하는 기법 2가지
- Count 기반으로 단어의 의미를 Vector화 → LSA, HAL
- 예측 기반으로 단어의 의미를 Vector화 → 전통 NNLM, RNNLM, Word2Vec, FastText 등
- Count 기반과 예측 기반 두 가지 방법을 모두 사용하는 방법 → Glove

---

# **Text Preprocessing**

## **Subword Segmentation**



# 단어 분리(Subword Segmentation)

- 현실적으로 이 세상의 모든 단어를 기계에서 학습하는 것은 불가능
- 단어 집합(Vocabulary)
  - 기계가 훈련 단계에서 학습한 단어들의 집합
  - 기계가 암기한 단어들의 리스트
- OOV(Out-Of-Vocabulary)
  - 테스트 단계에서 기계가 미처 배우지 못한 모르는 단어들
  - 단어 집합에 없는 단어
  - UNK(Unknown Word)으로 표현하기도
- 그렇다면 기계가 모르는 단어로 인해 문제를 풀지 못하는 상황을 OOV 문제라고 한다.

# 단어 분리(Subword Segmentation) (Cont.)

## ■ Subword Segmentation

- 기계가 아직 배운 적이 없는 단어더라도 대처할 수 있도록 도와주는 기법
- 기계 번역 등에서 주요 전처리로 사용되고 있다.

## ■ OOV 문제 해결 방법

- BPE(Byte Pair Encoding)
- WPM(Word Piece Model)

# BPE(Byte Pair Encoding) Algorithm

- 1994년에 데이터 압축 Algorithm으로 제안 된 후 자연어 처리의 단어 분리 Algorithm으로 응용.
- 기본적으로 연속적으로 가장 많이 등장한 글자의 쌍을 찾아서 하나의 글자로 병합하는 방식 수행

aaabdaaabac

- 위의 문자열 중 가장 자주 등장하고 있는 바이트의 쌍(byte pair)은 aa
- 이 aa라는 바이트의 쌍을 하나의 바이트인 Z로 치환.

aaabdaaabac → ZabdZabac | Z = aa

## BPE(Byte Pair Encoding) Algorithm (Cont.)

**aaabdaaabac** → **ZabdZabac** | **Z = aa**

- 이제 위 문자열 중에서 가장 많이 등장하고 있는 바이트의 쌍은 ab.

- 이제 이  $ab$ 를  $Y$ 로 치환.

$$ZabdZabac \rightarrow ZYdZYac \quad | \quad Z = aa \quad Y = ab$$

- 이제 가장 많이 등장하고 있는 바이트의 쌍은 ZY.

- ## ■ 이를 X로 치환.

**aaabdaaabac** → **ZabdZabac** → **ZYdZYac** → **XdXac**

**Z = aa**      **Y = ab**      **X = ZY**

# BPE(Byte Pair Encoding) Algorithm (Cont.)

- 단어 분리(Word Segmentation) Algorithm.
- 글자(Character) 단위에서 점차적으로 단어 집합(Vocabulary)을 만들어 내는 Bottom up 방식의 접근 사용.
- 훈련 데이터에 있는 단어들을 모든 글자(Characters) 또는 유니코드(Unicode) 단위로 단어 집합(Vocabulary) 생성
- 가장 많이 등장하는 Unigram을 하나의 Unigram으로 통합.

# 기존 접근 방법

```
# dictionary
```

```
# 훈련 데이터에 있는 단어와 등장 빈도수
```

```
low : 5, lower : 2, newest : 6, widest : 3
```

- 이 훈련 데이터에는 'low'란 단어가 5회 등장하였고, 'lower'란 단어는 2회 등장하였으며, 'newest'란 단어는 6회, 'widest'란 단어는 3회 등장했다는 의미.

```
# vocabulary
```

```
low, lower, newest, widest
```

- 현재의 이 훈련 데이터의 단어 집합에는 'low', 'lower', 'newest', 'widest'라는 4개의 단어가 존재.
- 만일 'lowest'란 단어가 등장하면 기계는 이 단어를 학습한 적이 없으므로 해당 단어에 대해서 제대로 대응하지 못하는 OOV 문제가 발생.

# BPE Algorithm 적용

```
# dictionary  
# 훈련 데이터에 있는 단어와 등장 빈도수  
low : 5, lower : 2, newest : 6, widest : 3
```

- 우선 dictionary의 모든 단어들을 글자(Character) 단위로 분리

```
# dictionary  
# 훈련 데이터에 있는 단어와 등장 빈도수  
l o w:5, l o w e r:2, n e w e s t:6, w i d e s t:3
```

- dictionary를 참고로 만든 초기 단어 집합(Vocabulary)
- 초기 구성은 글자 단위로 분리된 상태

```
# vocabulary  
l, o, w, e, r, n, s, t, i, d
```

## BPE Algorithm 적용 (Cont.)

- BPE Algorithm의 특징은 Algorithm의 동작을 몇 회 반복할 것인지를 사용자가 정해야 한다.
- 만일 총 10회를 수행한다고 가정하면, 즉, 가장 빈도수가 높은 Unigram의 쌍을 하나의 Unigram으로 통합하는 과정을 총 10회 반복한다는 것.
- 아래의 dictionary에 따르면 빈도수가 현재 가장 높은 Unigram의 쌍은 (e, s).

```
# dictionary
# 훈련 데이터에 있는 단어와 등장 빈도수
l o w:5, l o w e r:2, n e w e s t:6, w i d e s t:3
```

# BPE Algorithm 적용 (Cont.)

1회 - dictionary를 참고로 하였을 때 빈도수가 9로 가장 높은 (e, s)의 쌍을 es로 통합한다.

```
# dictionary update
```

```
| o w : 5,  
| o w e r : 2,  
n e w e s t : 6,  
w i d e s t : 3
```

```
# vocabulary update
```

```
|, o, w, e, r, n, s, t, i, d, es
```

# BPE Algorithm 적용 (Cont.)

2회 – 빈도가 9로 가장 높은 (es, t)의 쌍을 est로 통합한다.

```
# dictionary update
```

```
| o w : 5,  
| o w e r : 2,  
n e w est : 6,  
w i d est : 3
```

```
# vocabulary update
```

```
|, o, w, e, r, n, s, t, i, d, es, est
```

# BPE Algorithm 적용 (Cont.)

3회 – 빈도가 7로 가장 높은 (l, o)의 쌍을 lo로 통합한다.

```
# dictionary update
```

```
lo w:5,  
lo w e r:2,  
n e w est:6,  
w i d est:3
```

```
# vocabulary update
```

```
l, o, w, e, r, n, s, t, i, d, es, est, lo
```

# BPE Algorithm 적용 (Cont.)

## 10회째

```
# dictionary update
```

```
low : 5,  
low e r : 2,  
newest : 6,  
widest : 3
```

```
# vocabulary update
```

```
I, o, w, e, r, n, s, t, i, d, es, est, lo, low, ne, new, newest, wi, wid, widest
```

# BPE Algorithm 적용 (Cont.)

```
# vocabulary update  
l, o, w, e, r, n, s, t, i, d, es, est, lo, low, ne, new, newest, wi, wid, widest
```

- 이 경우 테스트 과정에서 *lowest*란 단어가 등장한다면
  - 기존에는 OOV에 해당되는 단어
  - BPE Algorithm을 사용한 위의 단어 집합에서는 더 이상 *lowest*는 OOV가 아니다.
  - 기계는 우선 *lowest*를 전부 글자 단위로 분할.  
*l, o, w, e, s, t*
  - 기계는 위의 단어 집합을 참고로 하여 *low*와 *est*를 찾는다.
  - *lowest*를 기계는 *low*와 *est* 두 단어로 encoding.
  - 이 두 단어는 둘 다 단어 집합에 있는 단어이므로 OOV가 아니다.

# BPE Algorithm 적용 (Cont.)

```
import re, collections  
  
def get_stats(vocab):  
    pairs = collections.defaultdict(int)  
    for word, freq in vocab.items():  
        symbols = word.split()  
        for i in range(len(symbols)-1):  
            pairs[symbols[i],symbols[i+1]] += freq  
    return pairs
```

```
def merge_vocab(pair, v_in):  
    v_out = {}  
    bigram = re.escape(' '.join(pair))  
    p = re.compile(r'(?<!WS)' + bigram + r'(?![WS])')  
    for word in v_in:  
        w_out = p.sub(" ".join(pair), word)  
        v_out[w_out] = v_in[word]  
    return v_out
```

```
vocab = {'l o w' : 5,  
         'l o w e r' : 2,  
         'n e w e s t': 6,  
         'w i d e s t': 3  
         }
```

```
num_merges = 10
```

```
for i in range(num_merges):  
    pairs = get_stats(vocab)  
    best = max(pairs, key=pairs.get)  
    vocab = merge_vocab(best, vocab)  
    print(best)
```

# BPE Algorithm 적용 (Cont.)

- BPE 알고리즘의 기본 원리는 가장 많이 등장한 문자열에 대하여 병합 하는 작업을 반복하는데, 원하는 단어 집합의 크기. 즉, 단어의 갯수가 될 때까지 이 작업을 반복하는 것이다.
- 단어 분리(Subword Segmentation)가 의미 있는 이유
  - 단어 분리(Subword Segmentation) 작업은 하나의 단어는 의미 있는 여러 단어들의 조합으로 구성된 경우가 많기 때문
  - 단어를 여러 단어로 분리해보겠다는 전처리 작업
  - 실제로, 언어의 특성에 따라 영어권 언어나 한국어는 단어 분리를 시도했을 때 어느 정도 의미 있는 단위로 나누는 것이 가능하다.

# WPM(Word Piece Model)

- 하나의 단어를 내부 단어(Subword Unit)들로 분리하는 단어 분리 모델.
- 2016년에 Google이 낸 논문(Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation [Wo et al. 2016])에서
- Google Translator에서 WPM이 어떻게 수행되는지에 대해 기술.

# WPM(Word Piece Model) (Cont.)

- WPM을 수행하기 이전의 문장
  - Jet makers feud over seat width with big orders at stake
- WPM을 수행한 결과(wordpieces)
  - \_J et \_makers \_fe ud \_over \_seat \_width \_with \_big \_orders \_at \_stake
- 기존에 존재하던 띄어쓰기는 underline(  )로 치환
- 단어는 내부단어(subword)로 통계에 기반하여 띄어쓰기로 분리

# WPM(Word Piece Model) (Cont.)

- 기존의 띄어쓰기를 underline(  )로 치환하는 이유는 차후 다시 문장 복원을 위한 장치이다.
- WPM의 결과로 나온 문장을 보면, 기존에 없던 띄어쓰기가 추가되어 내부 단어(subwords)들을 구분하는 구분자 역할을 수행.
- 본래의 띄어쓰기를 underline으로 치환해놓지 않으면, 기존 문장으로 복원할 수가 없다.
- WPM이 수행된 결과로부터 다시 수행 전의 결과로 돌리는 것 방법은 현재 있는 띄어쓰기를 전부 삭제하여 내부 단어들을 다시 하나의 단어로 연결시키고, underline을 다시 띄어쓰기로 바꾸면 된다.

# Language Model



# 언어 모델

- 언어라는 현상을 표현, 모델링하는 모델을 말한다.
- 기계가 자연어를 생성(Natural Language Generation, NLG) 하는 일들을 한다는 것을 의미.
- 언어 모델은 문장(단어 시퀀스)의 확률을 예측하는 모델.
- 이전 단어들이 주어졌을 때 다음 단어가 나올 확률 예측.
- 언어 모델은 단어들의 조합이 얼마나 적절한지, 또는 해당 문장이 얼마나 적합한지를 알려주는 일을 한다.
- 언어 모델을 만드는 방법
  - 통계를 이용한 방법
  - 인공 신경망을 이용한 방법

# 문장의 확률 예측

## ■ 기계 번역(Machine Translation):

$P(\text{나는 버스를 탔다}) > P(\text{나는 버스를 태운다})$

- 언어 모델은 두 문장을 비교하여 좌측의 문장의 확률이 더 높다고 판단.

## ■ 오타 교정(Spell Correction)

선생님이 교실로 부리나케

$P(\text{달려갔다}) > P(\text{잘려갔다})$

- 언어 모델은 두 문장을 비교하여 좌측의 문장의 확률이 더 높다고 판단.

## ■ 음성 인식(Speech Recognition)

$P(\text{나는 메롱을 먹는다}) < P(\text{나는 메론을 먹는다})$

- 언어 모델은 두 문장을 비교하여 우측의 문장의 확률이 더 높다고 판단.

## ■ 언어 모델은 위와 같이 확률을 통해 보다 적절한 문장을 판단한다.

# 언어 모델의 간단한 직관

## ■ Example

비행기를 타려고 공항에 갔는데 지각을 하는 바람에 비행기를 [?]

## ■ 人間 : '비행기를' 다음에 어떤 단어가 '놓쳤다'라고 예상.

- 人間의 지식에 기반하여 나올 수 있는 여러 단어들을 비교해본 결과 놓쳤다는 단어가 나올 확률이 가장 높다고 판단하였기 때.

## ■ Machine : ?

- 앞에 어떤 단어들이 나왔는지 고려하여 후보가 될 수 있는 여러 단어들에 대해서 확률을 예측해보고 가장 높은 확률을 가진 단어 선택.
- 이때 사용되는 것이 언어 모델이다.

## ■ 앞에 어떤 단어들이 나왔는지 고려하여 후보가 될 수 있는 여러 단어들에 대해서 확률을 예측해보고 가장 높은 확률을 가진 단어를 선택한다.

# 검색 엔진에서의 언어 모델의 예



딥 러닝을 이용한 |



- 딥 러닝을 이용한 부동산가격지수 예측
- 딥 러닝을 이용한 자연어 처리 입문
- 딥 러닝을 이용한 한국어 의존 구문 분석
- 딥 러닝을 이용한 개체명 인식
- 딥 러닝을 이용한 차량 번호판 검출
- 딥 러닝을 이용한 한국어 의미역 결정
- 딥 러닝을 이용한 한국어 형태소의 원형 복원 오류 수정
- 딥 러닝을 이용한
- 딥 러닝을 이용한 구문 분석

# 한국어에서의 언어 모델

- 한국어는 다음 단어를 예측하기 훨씬 어렵다.

- 한국어는 어순이 중요하지 않다.

나는 운동을 합니다. 체육관에서

나는 체육관에서 운동을 합니다.

체육관에서 운동을 합니다.

나는 운동을 체육관에서 합니다.

- 한국어는 교착어이다.

그녀 → 그녀가, 그녀를, 그녀의, 그녀와, 그녀로, 그녀께서, 그녀처럼

- 한국어는 띄어쓰기가 제대로 지켜지지 않는다.

# Count-based Word Representation





# **Count-based Word**

## **Representation**

### **다양한 단어의 표현 방법**

# 단어의 표현 방법

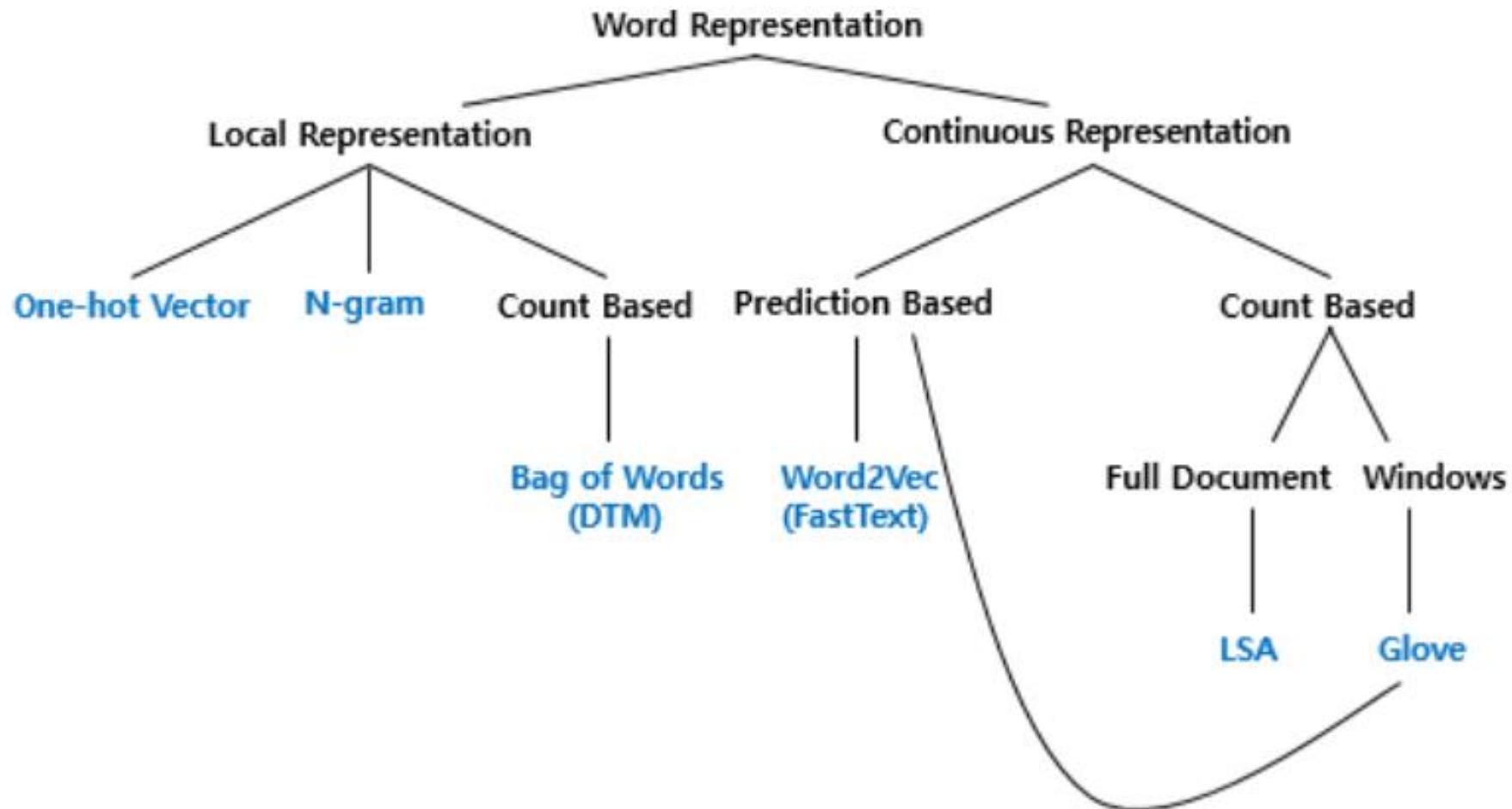
## ■ 국소 표현(Local representation) 방법

- 이산 표현(Discrete Representation)
- 해당 단어 그 자체만 보고, 특정값을 Mapping하여 단어를 표현하는 방법
- puppy(강아지), cute(귀여운), lovely(사랑스러운)라는 단어가 있을 때 각 단어에 1번, 2번, 3번 등과 같은 숫자를 Mapping하여 부여
- 단어의 의미, 뉘앙스 표현 불가

## ■ 분산 표현(Distributed Representation)

- 연속 표현(Continuous Representation)
- 그 단어를 표현하고자 주변을 참고하여 단어를 표현하는 방법.
- puppy(강아지)라는 단어 근처에는 주로 cute(귀여운), lovely(사랑스러운)이라는 단어가 자주 등장하므로, puppy라는 단어는 cute, lovely한 느낌이다로 단어 정의
- 단어의 의미, 뉘앙스 표현 가능

# 단어 표현의 Categories





---

# **Count-based Word Representation Bag of Words(BoW)**



# Bag of Words(BoW)

■ 단어들의 순서는 전혀 고려하지 않고, 단어들의 출현 빈도 (frequency)에만 집중하는 Text Data 수치화 표현 방법.

- ① 갖고 있는 어떤 Text 문서에 있는 단어들을 가방(Bag)에 전부 넣는다.
- ② 이 가방을 흔들어 단어들을 섞는다.
- ③ 만약, 해당 문서 내에서 특정 단어가 N번 등장했다면, 이 가방에는 그 특정 단어가 N개 있다.
- ④ 가방을 흔들어서 단어를 섞었기 때문에 더 이상 단어의 순서는 중요하지 않게 된다.

# Bag of Words(BoW) (Cont.)

## ■ BoW를 만드는 과정 요약

- ① 우선, 각 단어에 고유한 Index 부여.
- ② 각 Index의 위치에 단어 Token의 등장 횟수를 기록한 벡터 생성.

## ■ 한국어 예제

문서1 : 정부가 발표하는 물가상승률과 소비자가 느끼는 물가 상승률은 다르다.

- 입력된 문서에 대해서 단어 집합(vocabulary)을 만들어 Index를 할당, BoW를 만들기

# Bag of Words(BoW) (Cont.)

```
1 from konlpy.tag import Okt
2 import re
3 okt = Okt()
4
5 doc1 = "정부가 발표하는 물가상승률과 소비자가 느끼는 물가상승률은 다르다."
6 token = re.sub("(\\. )", "", doc1) # 정규 표현식을 통해 단공을 제거하는 정제(Cleaning) 작업.
7
8 token = okt.morphs(token) # MeCab 형태소 분석기를 통해 토큰화 작업을 수행.
9
10 word2index = {}
11 bow = []
12
13 for voca in token:
14     if voca not in word2index.keys():
15         word2index[voca] = len(word2index)
16         bow.insert(len(word2index)-1, 1) # BoW 전체에 전부 기본값 1을 넣는다. 단어의 개수는 최소 1개 이상이기 때문.
17     else:
18         index = word2index.get(voca)
19         bow[index] = bow[index] + 1 # 해당하는 단어의 Index를 받아온다.
20
21 print(word2index)
```

```
{'정부': 0, '가': 1, '발표': 2, '하는': 3, '물가상승률': 4, '과': 5, '소비자': 6, '느끼는': 7, '은': 8, '다르다': 9}
```

```
1 bow
```

```
[1, 2, 1, 1, 2, 1, 1, 1, 1, 1]
```

# Bag of Words(BoW) (Cont.)

- BoW에 있어서 중요한 것은 단어의 등장 빈도이다.
- 단어의 순서. 즉, Index의 순서는 전혀 상관없다.
- 다음과 같이 문서1에 대한 Index 할당을 임의로 바꾸고 그에 따른 BoW를 만든다고 가정해 보자.

```
# ('발표': 0, '가': 1, '정부': 2, '하는': 3, '소비자': 4, '과': 5, '물가상승률': 6, '느끼는': 7,  
'은': 8, '다르다': 9)  
[1, 2, 1, 1, 1, 2, 1, 1, 1]
```

- BoW는 단지 단어들의 Index만 바뀌었을 뿐이며, 개념적으로는 앞 Slide에서 만든 BoW와 동일한 BoW로 취급할 수 있다.

# Bag of Words(BoW) (Cont.)

- 문서2 : 소비자는 주로 소비하는 상품을 기준으로 물가상승률을 느낀다.

('소비자': 0, '는': 1, '주로': 2, '소비': 3, '하는': 4, '상품': 5, '을': 6, '기준': 7, '으로': 8, '물가상승률': 9, '느낀다': 10)  
[1, 1, 1, 1, 1, 2, 1, 1, 1, 1]

- 문서3: 정부가 발표하는 물가상승률과 소비자가 느끼는 물가상승률은 다르다. 소비자는 주로 소비하는 상품을 기준으로 물가상승률을 느낀다. (문서 1 + 문서 2)

('정부': 0, '가': 1, '발표': 2, '하는': 3, '물가상승률': 4, '과': 5, '소비자': 6, '느끼는': 7, '은': 8, '다르다': 9, '는': 10, '주로': 11, '소비': 12, '상품': 13, '을': 14, '기준': 15, '으로': 16, '느낀다': 17)  
[1, 2, 1, 2, 3, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1]

# Bag of Words(BoW) (Cont.)

- 문서3의 단어 집합은 문서1과 문서2의 단어들을 모두 포함하고 있다.
- BoW는 종종 여러 문서의 단어 집합을 합친 뒤에, 해당 단어 집합에 대한 각 문서의 BoW를 구하기도 한다.

```
문서3 단어 집합에 대한 문서1 BoW : [1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
문서3 단어 집합에 대한 문서2 BoW : [0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 2, 1, 1, 1]
```

- 물가상승률 :
  - 문서 3에서는 Index 4
  - 문서 1에서는 2번
  - 문서 2에서는 1번

Index 4

# Bag of Words(BoW) (Cont.)

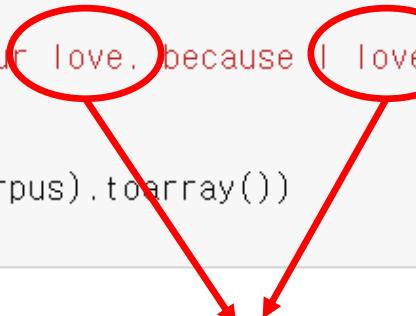
- BoW는 각 단어가 등장한 횟수를 수치화하는 Text 표현 방법
- 주로 어떤 단어가 얼마나 등장했는지를 기준으로 문서가 어떤 성격의 문서인지를 판단하는 작업에 사용.
  - 분류 문제
  - 여러 문서 간의 유사도를 구하는 문제
  - 예)'달리기', '체력', '근력'과 같은 단어가 자주 등장하면 해당 문서를 체육 관련 문서로 분류할 수 있다.
  - 예)'미분', '방정식', '부등식'과 같은 단어가 자주 등장한다면 수학 관련 문서로 분류할 수 있다.

# CountVectorizer Class로 BoW 만들기

- Scikit-learn에서 단어의 빈도를 Count하여 Vector로 만드는 Class.

```
1 from sklearn.feature_extraction.text import CountVectorizer  
2  
3 corpus = ['you know I want your love, because I love you.']  
4 vector = CountVectorizer()  
5  
6 print(vector.fit_transform(corpus).toarray())  
7 print(vector.vocabulary_)  
  
[[1 1 2 1 2 1]]  
{'you': 4, 'know': 1, 'want': 3, 'your': 5, 'love': 2, 'because': 0}
```

# Corpus로부터 각 단어의 빈도 수 기록.  
# 각 단어의 Index가 어떻게 부여되었는지 출력



- 알파벳 I는 BoW를 만드는 과정에서 사라졌는데, 이는 CountVectorizer가 기본적으로 길이가 2이상인 문자에 대해서만 토큰으로 인식하기 때문.

# CountVectorizer Class로 BoW 만들기 (Cont.)

## ■ CountVectorizer 사용시 주의할 점

- 띄어쓰기만을 기준으로 단어를 토큰화를 진행하고 BoW를 만든다는 점
- 영어인 경우는 문제 없지만, 한국어에 적용하면, 조사 등의 이유로 제대로 BoW가 만들어지지 않음을 의미.

## ■ 한국어사용시 CountVectorizer는 '물가상승률'이라는 단어를 인식하지 못함.

- '물가상승률과' 와 '물가상승률은' 으로 조사를 포함해서 하나의 단어로 판단
- '물가상승률과'와 '물가상승률은'이 각자 다른 인덱스에서 1이라는 빈도의 값을 갖게 된다.

# 불용어를 제거한 BoW 만들기

- CountVectorizer는 불용어를 지정하면, 불용어는 제외하고 BoW를 만들 수 있도록 불용어 제거 기능을 지원하고 있다.
- 사용자가 직접 정의한 불용어 사용하기

```
1 from sklearn.feature_extraction.text import CountVectorizer  
2  
3 text = ["Family is not an important thing. It's everything."]  
4 vect = CountVectorizer(stop_words=["the", "a", "an", "is", "not"])  
5  
6 print(vect.fit_transform(text).toarray())  
7 print(vect.vocabulary_)
```

```
[[1 1 1 1 1]]  
{'family': 1, 'important': 2, 'thing': 4, 'it': 3, 'everything': 0}
```

# 불용어를 제거한 BoW 만들기 (Cont.)

## ■ CountVectorizer에서 제공하는 자체 불용어 사용하기

```
1 from sklearn.feature_extraction.text import CountVectorizer  
2  
3 text = ["Family is not an important thing. It's everything."]  
4 vect = CountVectorizer(stop_words="english")  
5  
6 print(vect.fit_transform(text).toarray())  
7 print(vect.vocabulary_)
```

```
[[1 1 1]]  
{'family': 0, 'important': 1, 'thing': 2}
```

# 불용어를 제거한 BoW 만들기 (Cont.)

## ■ NLTK에서 지원하는 불용어 사용하기

```
1 from sklearn.feature_extraction.text import CountVectorizer  
2 from nltk.corpus import stopwords  
3  
4 text = ["Family is not an important thing. It's everything."]  
5 sw = stopwords.words("english")  
6 vect = CountVectorizer(stop_words = sw)  
7 print(vect.fit_transform(text).toarray())  
8 print(vect.vocabulary_)
```

```
[[1 1 1 1]]  
{'family': 1, 'important': 2, 'thing': 3, 'everything': 0}
```



# **Count-based Word Representation**

## **DTM**



# 문서 단어 행렬(Document-Term Matrix, DTM)

## ■ 문서 단어 행렬(Document-Term Matrix, DTM)

- 다수의 문서에서 등장하는 각 단어들의 빈도를 행렬로 표현한 것.
- 각 문서에 대한 BoW를 하나의 행렬로 만든 것.
- BoW 표현을 다수의 문서에 대해서 행렬로 표현하고 부르는 용어

## ■ 문서 단어 행렬(Document-Term Matrix, DTM)의 표기법

- 문서 단어 행렬(Document-Term Matrix, DTM)이란 다수의 문서에서 등장하는 각 단어들의 빈도를 행렬로 표현한 것을 말합니다. 쉽게 생각하면 각 문서에 대한 BoW를 하나의 행렬로 만든 것으로 생각할 수 있으며, BoW와 다른 표현 방법이 아니라 BoW 표현을 다수의 문서에 대해서 행렬로 표현하고 부르는 용어입니다. 예를 들어서 이렇게 4개의 문서가 있다고 합시다.

# 문서 단어 행렬(Document-Term Matrix, DTM) (Cont.)

## ■ 문서 단어 행렬(Document-Term Matrix, DTM)의 표기법

- 문서1 : 먹고 싶은 사과
- 문서2 : 먹고 싶은 바나나
- 문서3 : 길고 노란 바나나 바나나
- 문서4 : 저는 과일이 좋아요

# 문서 단어 행렬(Document-Term Matrix, DTM) (Cont.)

## ■ 문서 단어 행렬(Document-Term Matrix, DTM)의 표기법

# 문서 단어 행렬(Document-Term Matrix, DTM) (Cont.)

## ■ DTM의 한계

- 희소 표현(Sparse representation)
- 단순 빈도 수 기반 접근

# 문서 단어 행렬(Document-Term Matrix, DTM) (Cont.)

## ■ DTM의 한계 : 희소 표현(Sparse representation)

- DTM에서 각 문서를 문서 Vector라고 한다면,
- One-hot Vector처럼 DTM도 각 문서 벡터의 차원은 전체 단어 집합의 크기를 가지게 된다.
- 가지고 있는 전체 Corpus가 방대한 데이터라면 문서 벡터의 차원은 수백만의 차원을 가질 수도 있다.
- 많은 문서 벡터가 대부분의 값이 0을 가질 수는 희소 벡터(Sparse Vector)가 될 수 있다.

## ■ 대책

- 전처리를 통해 단어 집합의 크기를 줄이는 일이 필요
- 구두점, 빈도수가 낮은 단어, 불용어를 제거하고, 어간이나 표제어 추출을 통해 단어를 정규화하여 단어 집합의 크기를 줄여야 한다.

# 문서 단어 행렬(Document-Term Matrix, DTM) (Cont.)

## ■ DTM의 한계 : 단순 빈도 수 기반 접근

- 여러 문서에 등장하는 모든 단어에 대해서 빈도 표기를 하는 방법의 문제점
- 불용어들은 자주 등장하게 마련.

## ■ 대책

- DTM에 불용어와 중요한 단어에 대해서 가중치를 줄 수 있는 방법이 필요 → TF-IDF



---

# **Count-based Word Representation**

## **TF-IDF**



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

- 단어의 빈도와 역 문서 빈도(문서의 빈도에 특정 식을 취함)를 사용하여 DTM 내의 각 단어들마다 중요한 정도를 가중치로 주는 방법.
- TF와 IDF를 곱한 값을 의미.
- DTM을 만든 후에, 거기에 TF-IDF 가중치를 주면 된다.
- 사용처
  - 주로 문서의 유사도를 구하는 작업
  - 검색 시스템에서 검색 결과의 중요도를 정하는 작업
  - 문서 내에서 특정 단어의 중요도를 구하는 작업 등

# TF-IDF(Term Frequency-Inverse Document Frequency) (Cont.)

■ 문서를  $d$ , 단어를  $t$ , 문서의 총 개수를  $n$ 이라고 표현할 때 TF, DF, IDF는 각각 다음과 같이 정의할 수 있다.

- 1)  $tf(d, t)$  : 특정 문서  $d$ 에서의 특정 단어  $t$ 의 등장 횟수.
- 2)  $df(t)$  : 특정 단어  $t$ 가 등장한 문서의 수.
- 3)  $idf(d, t)$  :  $df(t)$ 에 반비례하는 수.

$$idf(d, t) = \log\left(\frac{n}{1 + df(t)}\right)$$

※  $\log$ 를 사용하지 않으면 IDF를 DF의 역수로 사용했을 때, 총 문서의 수  $n$ 이 커질 수록, IDF의 값이 기하급수적으로 커지기 위함을 방지.

※ 분모에 1을 더하는 이유는 특정 단어가 전체 문서에서 등장하지 않을 경우 분모가 0이 되는 것을 막기 위해서.

# TF-IDF(Term Frequency-Inverse Document Frequency) (Cont.)

log 사용할 때

$$idf(d, t) = \log(n/df(t))$$

$$n = 1,000,000$$

단어 $t$	$df(t)$	$idf(d, t)$
word1	1	6
word2	100	4
word3	1,000	3
word4	10,000	2
word5	100,000	1
word6	1,000,000	0

log 사용하지 않을 때

$$idf(d, t) = n/df(t)$$

$$n = 1,000,000$$

단어 $t$	$df(t)$	$idf(d, t)$
word1	1	1,000,000
word2	100	10,000
word3	1,000	1,000
word4	10,000	100
word5	100,000	10
word6	1,000,000	1

# TF-IDF(Term Frequency-Inverse Document Frequency) (Cont.)

## ■ 중요도 파악

- 모든 문서에서 자주 등장하는 단어는 중요도가 낮다고 판단
- 특정 문서에서만 자주 등장하는 단어는 중요도가 높다고 판단
- TF-IDF 값이 낮으면 중요도가 낮은 것
- TF-IDF 값이 크면 중요도가 큰 것

## ■ 불용어의 경우에는 모든 문서에 자주 등장하기 마련이기 때문에 다른 단어의 TF-IDF에 비해서 낮아지게 된다.

# TF-IDF(Term Frequency-Inverse Document Frequency) (Cont.)

- 아래의 예제를 가지고 TF-IDF의 값을 구해보자.
  - 위에서 사용한 DTM이 곧 각 문서에서 각 단어의 TF이다.

# TF-IDF(Term Frequency-Inverse Document Frequency) (Cont.)

## ■ TF와 곱할 IDF의 값

단어	IDF(역 문서 빈도)
과일이	$\ln(4/(1+1)) = 0.693147$
길고	$\ln(4/(1+1)) = 0.693147$
노란	$\ln(4/(1+1)) = 0.693147$
먹고	$\ln(4/(2+1)) = 0.287682$
바나나	$\ln(4/(2+1)) = 0.287682$
사과	$\ln(4/(1+1)) = 0.693147$
싶은	$\ln(4/(2+1)) = 0.287682$
저는	$\ln(4/(1+1)) = 0.693147$
좋아요	$\ln(4/(1+1)) = 0.693147$

# TF-IDF(Term Frequency-Inverse Document Frequency) (Cont.)

## ■ TF-IDF의 값

-	과일이	길고	노란	먹고	바나나	사과	싫은	저는	좋아요
문서1	0	0	0	0.287682	0	0.693147	0.287682	0	0
문서2	0	0	0	0.287682	0.287682	0	0.287682	0	0
문서3	0	0.693147	0.693147	0	0.575364	0	0	0	0
문서4	0.693147	0	0	0	0	0	0	0.693147	0.693147

# Scikit-learn을 이용한 DTM과 TF-IDF 실습

- DTM 또한 BoW 행렬이기 때문에, 앞서 BoW에서 배운 **CountVectorizer**를 사용하면 간단히 DTM을 만들 수 있다.

```
1 from sklearn.feature_extraction.text import CountVectorizer
2
3 corpus = [
4     'you know I want your love',
5     'I like you',
6     'what should I do ',
7 ]
8
9 vector = CountVectorizer()
10 print(vector.fit_transform(corpus).toarray())      # Corpus로부터 각 단어의 빈도 수를 기록한다.
11 print(vector.vocabulary_)                         # 각 단어의 인덱스가 어떻게 부여되었는지를 보여준다.
```

```
[[0 1 0 1 0 1 0 1 1]
 [0 0 1 0 0 0 0 1 0]
 [1 0 0 0 1 0 1 0 0]]
{'you': 7, 'know': 1, 'want': 5, 'your': 8, 'love': 3, 'like': 2, 'what': 6, 'should': 4, 'do': 0}
```

# Scikit-learn을 이용한 DTM과 TF-IDF 실습 (Cont.)

## ■ TfidfVectorizer

- Scikit-learn에서 TF-IDF를 자동 계산해주는 Class

```
1 from sklearn.feature_extraction.text import TfidfVectorizer  
2  
3 corpus = [  
4     'you know I want your love',  
5     'I like you',  
6     'what should I do ',  
7 ]  
8  
9 tfidfv = TfidfVectorizer().fit(corpus)  
10 print(tfidfv.transform(corpus).toarray())  
11 print(tfidfv.vocabulary_)
```

```
[[0.          0.46735098 0.          0.46735098 0.          0.46735098  
 0.          0.35543247 0.46735098]  
[0.          0.          0.79596054 0.          0.          0.  
 0.          0.60534851 0.          ]]  
[0.57735027 0.          0.          0.          0.57735027 0.  
 0.57735027 0.          0.          ]]  
{'you': 7, 'know': 1, 'want': 5, 'your': 8, 'love': 3, 'like': 2, 'what': 6, 'should': 4, 'do': 0}
```

# Document Similarity



# Document Similarity



## 문서 유사도

# 문서 유사도(Document Similarity)

- 문서의 유사도를 구하는 일은 자연어 처리의 주요 주제 중 하나
- 인간
  - 문서들 간에 동일한 단어 또는 비슷한 단어가 얼마나 공통적으로 많이 사용되었는가?
- Machine
  - 문서의 유사도의 성능은 각 문서의 단어들을 어떤 방법으로 수치화하여 표현했는지(DTM, Word2Vec 등), 문서 간의 단어들의 차이를 어떤 방법(유clidean 거리, 코사인 유사도 등)으로 계산했는가?

# Document Similarity



## Cosine Similarity

# Cosine Similarity

- 두 벡터 간의 Cosine 각도를 이용하여 구할 수 있는 두 벡터의 유사도를 의미
- 두 벡터의 방향이 완전히 동일한 경우에는 1의 값
- 두 벡터의 방향이  $90^\circ$ 의 각을 이루면 0
- 두 벡터의 방향이  $180^\circ$ 로 반대의 방향을 가지면 -1
- Cosine 유사도는 -1 이상 1 이하의 값을 가지며 값이 1에 가까울수록 유사도가 높다고 판단할 수 있다.
- 즉, 두 벡터가 가리키는 방향이 얼마나 유사한가를 의미



# Cosine Similarity (Cont.)

- 두 Vector A, B에 대해서 Cosine 유사도를 식으로 표현

$$similarity = \cos(\Theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n (A_i)^2} \times \sqrt{\sum_{i=1}^n (B_i)^2}}$$

# Cosine Similarity (Cont.)

- DTM이나 TF-IDF 행렬을 통해서 문서의 유사도를 구하는 경우에는 DTM이나 TF-IDF 행렬이 각각의 특징 벡터 A, B가 된다.
- 문서1 : 저는 사과 좋아요
- 문서2 : 저는 바나나 좋아요
- 문서3 : 저는 바나나 좋아요 저는 바나나 좋아요

- DTM

-	바나나	사과	저는	좋아요
문서1	0	1	1	1
문서2	1	0	1	1
문서3	2	0	2	2

# Cosine Similarity (Cont.)

```
| 1 from numpy import dot  
| 2 from numpy.linalg import norm  
| 3 import numpy as np  
| 4  
| 5 def cos_sim(A, B):  
| 6     return dot(A, B)/(norm(A)*norm(B))
```

```
| 1 doc1 = np.array([0,1,1,1])  
| 2 doc2 = np.array([1,0,1,1])  
| 3 doc3 = np.array([2,0,2,2])
```

```
| 1 print(cos_sim(doc1, doc2)) #문서1과 문서2의 코사인 유사도  
| 2 print(cos_sim(doc1, doc3)) #문서1과 문서3의 코사인 유사도  
| 3 print(cos_sim(doc2, doc3)) #문서2과 문서3의 코사인 유사도
```

0.6666666666666667

0.6666666666666667

1.0000000000000002

# 유사도를 이용한 추천 시스템 구현하기

## ■ 준비

- Kaggle에서 사용되었던 영화 Dataset을 가지고 영화 추천 시스템을 만들어 본다.
- TF-IDF와 Cosine 유사도만으로 영화의 줄거리에 기반해서 영화를 추천하는 추천 시스템을 만들 수 있다.
- 다운로드 <https://www.kaggle.com/rounakbanik/the-movies-dataset>
- 원본 파일은 위 링크에서 movies\_metadata.csv 파일 다운로드
- 해당 데이터는 총 24개의 열을 가진 45,466개의 샘플로 구성된 영화 정보 데이터이다.

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

## ■ 준비

```
1 import pandas as pd  
2 data = pd.read_csv('movies_metadata.csv', low_memory=False)  
3  
4 data.head(2)
```

	adult	belongs_to_collection	budget	genres	homepage	id	imdb_id	original_language	original_title	overview	...	release_
0	False	{'id': 10194, 'name': 'Toy Story Collection', ...}	30000000	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...']}	http://toystory.disney.com/toy-story	862	tt0114709	en	Toy Story	Led by Woody, Andy's toys live happily in his ...	...	1995-
1	False		NaN	65000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...']}	NaN	8844	tt0113497	Jumanji	When siblings Judy and Peter discover an encha...	...	1995-

2 rows × 24 columns

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

- 여기서 Cosine 유사도에 사용할 데이터는 영화 제목에 해당하는 title 열과 줄거리에 해당하는 overview 열 이다.
- 좋아하는 영화를 입력하면, 해당 영화의 줄거리와 줄거리가 유사한 영화를 찾아서 추천하는 시스템을 만들 것이다.
- 먼저 20,000개의 Sample
- TF-IDF할 때 Data에 Null이 있으면 Error 발생함.
- TF-IDF의 대상이 되는 Data의 Overview 열에 Null이 있는지 여부 파악

```
1 data = data.head(20000)
```

```
1 data[ 'overview' ].isnull().sum()
```

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

- 135개의 샘플에서 Null 값이 있다.
- Null을 제거하기 위해 Pandas를 이용하여 Null값을 처리하는 도구인 fillna()를 사용한다.
- Null값을 빈 값(empty value)으로 대체하여 Null 값을 제거 한다.

```
1 data['overview'] = data['overview'].fillna('')  
2 # overview에서 Null 값을 가진 경우에는 빈 제거
```

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

## ■ Null 값 제거후, tf-idf 수행

```
1 from sklearn.feature_extraction.text import TfidfVectorizer  
2  
3 tfidf = TfidfVectorizer(stop_words='english')  
4 tfidf_matrix = tfidf.fit_transform(data['overview']) # overview에 대해서 tf-idf 수행  
5 print(tfidf_matrix.shape)
```

(20000, 47487)

## ■ 20,000개의 영화를 표현하기위해 총 47,487개의 단어가 사용되었음을 보여주고 있다.

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

- Cosine 유사도를 사용해서 문서의 유사도를 구할 수 있다.

```
1 from sklearn.metrics.pairwise import linear_kernel  
2  
3 cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

- 영화의 타이틀과 Index를 가진 테이블을 만들고 이 중 5개만 출력해본다.

```
1 indices = pd.Series(data.index, index=data['title']).drop_duplicates()  
2 print(indices.head())
```

```
title  
Toy Story          0  
Jumanji           1  
Grumpier Old Men 2  
Waiting to Exhale 3  
Father of the Bride Part II 4  
dtype: int64
```

- 영화의 타이틀을 입력하면 Index를 리턴한다.

```
1 idx = indices['Father of the Bride Part II']  
2 print(idx)
```

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

- 이제 선택한 영화에 대해서 Cosine 유사도를 이용하여, 가장 overview가 유사한 10개의 영화를 찾아내는 함수를 만든다.

```
1 def get_recommendations(title, cosine_sim=cosine_sim):
2     # 선택한 영화의 타이틀로부터 해당되는 Index를 받아온다.
3     # 이제 선택한 영화를 가지고 연산할 수 있다.
4     idx = indices[title]
5
6     # 모든 영화에 대해서 해당 영화와의 유사도를 구한다.
7     sim_scores = list(enumerate(cosine_sim[idx]))
8
9     # 유사도에 따라 영화들을 정렬한다.
10    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
11
12    # 가장 유사한 10개의 영화를 받아온다.
13    sim_scores = sim_scores[1:11]
14
15    # 가장 유사한 10개의 영화의 Index를 받아온다.
16    movie_indices = [i[0] for i in sim_scores]
17
18    # 가장 유사한 10개의 영화의 제목을 리턴한다.
19    return data['title'].iloc[movie_indices]
```

# 유사도를 이용한 추천 시스템 구현하기 (Cont.)

- 영화 [다크 나이트 라이즈]와 overview가 유사한 영화들을 찾아본다.

```
1 get_recommendations('The Dark Knight Rises')
```

```
12481           The Dark Knight
150             Batman Forever
1328            Batman Returns
15511           Batman: Under the Red Hood
585             Batman
9230            Batman Beyond: Return of the Joker
18035           Batman: Year One
19792           Batman: The Dark Knight Returns, Part 1
3095            Batman: Mask of the Phantasm
10122           Batman Begins
Name: title, dtype: object
```

# Document Similarity

여러가지 유사도 기법



# 유클리드 거리(Euclidean Distance)

- 잘 알고 있는 거리 공식(=유클리디언 거리)을 이용하여 계산
- 계산 값이 0에 가까울수록 유사.
- n차원의 tuple에 대해서 유클리디언 거리는 다음과 같이 계산한다.

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

where  $x = (x_1, x_2, \dots, x_n)$ ,  $y = (y_1, y_2, \dots, y_n)$

# 유클리드 거리(Euclidean Distance) (Cont.)

- 아래와 같은 DTM이 있을 때

-	바나나	사과	저는	좋아요
문서1	2	3	0	1
문서2	1	2	3	1
문서3	2	1	2	2

- 다음과 같은 문서Q에 대해서 문서1, 문서2, 문서3 중 가장 유사한 문서를 찾아내려면.

-	바나나	사과	저는	좋아요
문서Q	1	1	0	1

# 유클리드 거리(Euclidean Distance) (Cont.)

- 유클리드 거리의 값이 가장 작다는 것은, 문서 간의 거리가 가장 가깝다는 것을 의미
- 즉, 문서1이 문서Q와 가장 유사하다고 볼 수 있다.

```
1 import numpy as np  
2  
3 def dist(x,y):  
4     return np.sqrt(np.sum((x-y)**2))
```

```
1 doc1 = np.array((2,3,0,1))  
2 doc2 = np.array((1,2,3,1))  
3 doc3 = np.array((2,1,2,2))  
4 docQ = np.array((1,1,0,1))  
5  
6 print(dist(doc1,docQ))  
7 print(dist(doc2,docQ))  
8 print(dist(doc3,docQ))
```

2.23606797749979  
3.1622776601683795  
2.449489742783178

# 자카드 유사도(Jaccard Similarity)

- A와 B 두 개의 집합이 있다.
- 이때 교집합은 두 개의 집합에서 공통으로 가지고 있는 원소들의 집합을 말한다.
- 즉, 합집합에서 교집합의 비율을 구한다면 두 집합 A와 B의 유사도를 구할 수 있다는 것
- 0과 1사이의 값을 가진다.
- 만약 두 집합이 동일하다면 1의 값을 가지고, 두 집합의 공통 원소가 없다면 0의 값을 갖는다.

# 자카드 유사도(Jaccard Similarity) (Cont.)

- 자카드 유사도를 구하는 함수를  $J$ 라고 하고, 자카드 유사도 함수  $J$ 는 아래와 같다.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

- 두 개의 비교할 문서를 각각  $doc1$ ,  $doc2$ 라고 했을 때  $doc1$ 과  $doc2$ 의 문서의 유사도를 구하기 위한 자카드 유사도는 아래와 같다.

$$J(doc_1, doc_2) = \frac{doc_1 \cap doc_2}{doc_1 \cup doc_2}$$

- 즉, 두 문서  $doc1$ ,  $doc2$  사이의 자카드 유사도  $J(doc1, doc2)$ 는 두 집합의 교집합 크기를 두 집합의 합집합 크기로 나눈 값으로 정의된다.

# 자카드 유사도(Jaccard Similarity) (Cont.)

## ■ 다음의 예제를 보자.

```
1 # 다음과 같은 두 개의 문서가 있다.
2 # 두 문서 모두에서 등장한 단어는 apple과 banana 2개.
3 doc1 = "apple banana everyone like likey watch card holder"
4 doc2 = "apple banana coupon passport love you"
5
6 # 토큰화를 수행한다.
7 tokenized_doc1 = doc1.split()
8 tokenized_doc2 = doc2.split()
9
10 # 토큰화 결과 출력한다.
11 print(tokenized_doc1)
12 print(tokenized_doc2)
```

```
['apple', 'banana', 'everyone', 'like', 'likey', 'watch', 'card', 'holder']
['apple', 'banana', 'coupon', 'passport', 'love', 'you']
```

# 자카드 유사도(Jaccard Similarity) (Cont.)

- 문서 1과 문서 2의 합집합을 구한다.

```
1 union = set(tokenized_doc1).union(set(tokenized_doc2))  
2 print(union)
```

```
{'apple', 'like', 'card', 'coupon', 'passport', 'banana', 'holder', 'love', 'watch', 'everyone', 'you', 'likey'}
```

- 문서1과 문서2의 합집합의 단어의 총 개수는 12개이다.
- 문서1과 문서2의 교집합을 구한다.

```
1 intersection = set(tokenized_doc1).intersection(set(tokenized_doc2))  
2 print(intersection)
```

```
{'apple', 'banana'}
```

# 자카드 유사도(Jaccard Similarity) (Cont.)

- 문서1과 문서2에서 둘 다 등장한 단어는 banana와 apple 총 2개이다.
- 교집합의 수를 합집합의 수로 나누면 자카드 유사도가 계산된다.

```
1 print(len(intersection)/len(union)) # 2를 12로 나눔.
```

0.1666666666666666

- 위의 값은 자카드 유사도이자, 두 문서의 총 단어 집합에서 두 문서에서 공통적으로 등장한 단어의 비율이기도 하다.

---

# Topic Modeling





---



# **Topic Modeling**

## **LSA(Latent Semantic Analysis)**

# Topic Modeling

- 토픽(Topic)은 한국어로는 주제라고 한다.
- 토픽 모델링(Topic Modeling)이란 기계 학습 및 자연어 처리 분야에서 토픽이라는 문서 집합의 추상적인 주제를 발견하기 위한 통계적 모델 중 하나이다.
- 텍스트 본문의 숨겨진 의미 구조를 발견하기 위해 사용되는 텍스트 마이닝 기법이다.

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- LSA는 정확히는 토픽 모델링을 위해 최적화 된 알고리즘은 아니지만, 토픽 모델링이라는 분야에 아이디어를 제공한 알고리즘이라고 볼 수 있다.
- 뒤에서 배우게 되는 LDA는 LSA의 단점을 개선하여 탄생한 알고리즘으로 토픽 모델링에 보다 적합한 알고리즈다.
- BoW에 기반한 DTM이나 TF-IDF는 기본적으로 단어의 빈도 수를 이용한 수치화 방법이기 때문에 단어의 의미를 고려하지 못한다는 단점이 있다.
- 이를 위한 대안으로 DTM의 잠재된(Latent) 의미를 이끌어내는 방법으로 잠재 의미 분석(Latent Semantic Analysis, LSA)이라는 방법이 있다.
- 잠재 의미 분석(Latent Semantic Indexing, LSI)이라고 부르기도 한다.

# 특이값 분해(Singular Value Decomposition, SVD)

- $A$ 가  $m \times n$  행렬일 때, 다음과 같이 3개의 행렬의 곱으로 분해(decomposition)하는 것을 말한다.

$$A = U\Sigma V^T$$

- 여기서 각 3개의 행렬은 다음과 같은 조건을 만족한다.

$U : m \times m$  직교행렬 ( $AA^T = U(\Sigma\Sigma^T)U^T$ )

$V : n \times n$  직교행렬 ( $A^T A = V(\Sigma^T\Sigma)V^T$ )

$\Sigma : m \times n$  직사각 대각행렬

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

- 여기서 직교행렬(orthogonal matrix)이란 자신과 자신의 전치 행렬(transposed matrix)의 곱 또는 이를 반대로 곱한 결과가 단위행렬(identity matrix)이 되는 행렬을 말한다.
- 대각행렬(diagonal matrix)이란 주대각선을 제외한 곳의 원소가 모두 0인 행렬을 의미한다.
- 이때 SVD로 나온 대각 행렬의 대각 원소의 값을 행렬 A의 특이값(singular value)라고 한다.

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

## 전치 행렬(Transposed Matrix)

- 원래의 행렬에서 행과 열을 바꾼 행렬이다.
- 주대각선을 축으로 반사 대칭을 하여 얹는 행렬이다.
- 기호는 기존 행렬 표현의 우측 위에 T를 붙인다.
- 예)기존의 행렬을 M이라고 한다면, 전치 행렬은  $M^T$ 와 같이 표현한다.

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad M^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

## 단위 행렬(Identity Matrix)

- 주대각선의 원소가 모두 1이며 나머지 원소는 모두 0인 정사각 행렬이다.
- 보통 줄여서 대문자  $I$ 로 표현하기도 하는데,  $2 \times 2$  단위 행렬과  $3 \times 3$  단위 행렬을 표현해보면 다음과 같다.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

## 역 행렬(Inverse Matrix)

- 만약 행렬  $A$ 와 어떤 행렬을 곱했을 때, 결과로서 단위 행렬이 나온다면 이때의 어떤 행렬을  $A$ 의 역행렬이라고 한다.
- $A^{-1}$ 라고 표현한다.

$$A \times A^{-1} = I$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} ? & & \\ & ? & \\ & & ? \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

## 직교 행렬(Orthogonal Matrix)

- 실수  $n \times n$  행렬  $A$ 에 대해서  $A \times A^T = I$  를 만족하면서  $A^T \times A = I$  을 만족하는 행렬  $A$ 를 직교 행렬이라고 한다.
- 그런데 역행렬의 정의를 다시 생각해보면, 결국 직교 행렬은  $A^{-1} = A^T$ 를 만족한다.

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

## 대각 행렬(Diagonal Matrix)

- 주대각선을 제외한 곳의 원소가 모두 0인 행렬이다.
- 아래의 그림에서는 주대각선의 원소를  $a$ 라고 표현하고 있다.
- 만약 대각 행렬  $\Sigma$ 가  $3 \times 3$  행렬이라면, 다음과 같은 모양을 가진다.

$$\Sigma = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix}$$

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

## 대각 행렬(Diagonal Matrix)

- 지금까지는 정사각 행렬이기 때문에 직관적으로 이해가 쉽다.
- 직사각 행렬이 될 경우를 잘 보아야 헷갈리지 않는다.
- 만약 행의 크기가 열의 크기보다 크다면 다음과 같은 모양을 가진다.
- 즉,  $m \times n$  행렬일 때,

$$\Sigma = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \\ 0 & 0 & 0 \end{bmatrix} \quad m > n \text{ 인 경우} \quad \Sigma = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \end{bmatrix} \quad n > m \text{ 인 경우}$$

# 특이값 분해(Singular Value Decomposition, SVD) (Cont.)

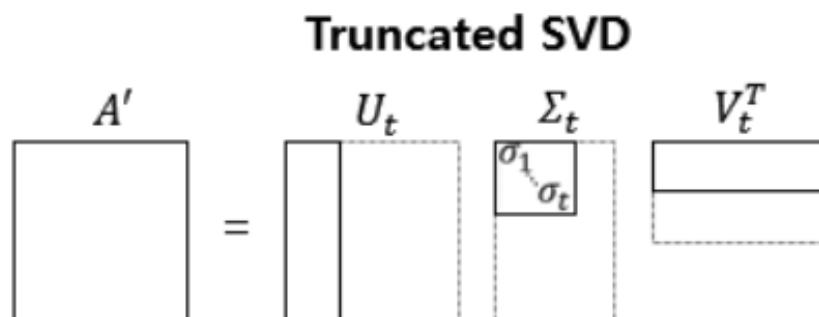
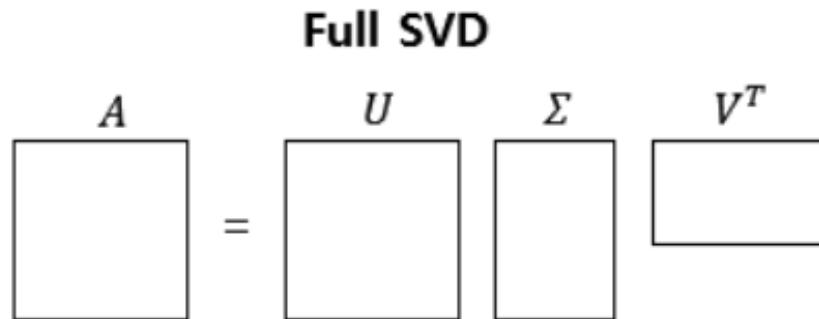
## 대각 행렬(Diagonal Matrix)

- SVD를 통해 나온 대각 행렬  $\Sigma$ 는 추가적인 성질을 가지는데, 대각 행렬  $\Sigma$ 의 주대각원소를 행렬 A의 특이값(singular value)라고 하며, 이를  $\sigma_1, \sigma_2, \dots, \sigma_r$ 라고 표현한다고 하였을 때 특이값  $\sigma_1, \sigma_2, \dots, \sigma_r$ 은 내림차순으로 정렬되어 있다는 특징을 가진다.
- 아래의 그림은 특이값 12.4, 9.5, 1.3이 내림차순으로 정렬되어져 있는 모습을 보여준다.

$$\Sigma = \begin{bmatrix} 12.4 & 0 & 0 \\ 0 & 9.5 & 0 \\ 0 & 0 & 1.3 \end{bmatrix}$$

# 절단된 SVD(Truncated SVD)

- 이제까지 설명한 SVD를 풀 SVD(full SVD)라고 한다.
- 하지만 LSA의 경우 풀 SVD에서 나온 3개의 행렬에서 일부 Vector들을 삭제시킨 절단된 SVD(truncated SVD)를 사용한다.



## 절단된 SVD(Truncated SVD) (Cont.)

- 앞의 그림을 보면, 절단된 SVD는 대각 행렬  $\Sigma$ 의 대각 원소의 값 중에서 상위값  $t$ 개만 남게 된다.
- 절단된 SVD를 수행하면 값의 손실이 일어나므로 기존의 행렬  $A$ 를 복구할 수 없다.
- 또한,  $U$ 행렬과  $V$ 행렬의  $t$ 열까지만 남긴다.
- 여기서  $t$ 는 우리가 찾고자 하는 Topic의 수를 반영한 Hyper-Parameter값이다.
- Hyper-Parameter란 사용자가 직접 값을 선택하며 성능에 영향을 주는 매개변수를 말한다.
- $t$ 를 선택하는 것은 쉽지 않은 일이다.
- $t$ 를 크게 잡으면 기존의 행렬  $A$ 로부터 다양한 의미를 가져갈 수 있지만,  $t$ 를 작게 잡아야만 Noise를 제거할 수 있기 때문이다.

# 절단된 SVD(Truncated SVD) (Cont.)

- 이렇게 일부 벡터들을 삭제하는 것을 데이터의 차원을 줄인다고도 말하는데, 데이터의 차원을 줄이게 되면 당연히 Full SVD를 하였을 때보다 직관적으로 계산 비용이 낮아지는 효과를 얻을 수 있다.
- 또한, 계산 비용이 낮아지는 것 외에도 상대적으로 중요하지 않은 정보를 삭제하는 효과를 갖고 있다.
- 이것을 영상 처리 분야에서는 Noise를 제거한다고 한다.
- 자연어 처리 분야에서는 설명력이 낮은 정보를 삭제하고 설명력이 높은 정보를 남긴다는 의미를 갖는다.
- 즉, 기존의 행렬에서는 드러나지 않았던 심층적인 의미를 확인할 수 있게 해준다.

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 기존의 DTM이나 DTM에 단어의 중요도에 따른 가중치를 주었던 TF-IDF 행렬은 단어의 의미를 전혀 고려하지 못한다는 단점을 갖고 있었다.
- LSA는 기본적으로 DTM이나 TF-IDF 행렬에 절단된 SVD(Truncated SVD)를 사용하여 차원을 축소시키고, 단어들의 잠재적인 의미를 끌어낸다는 아이디어를 갖고 있다.

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

-	과일이	길고	노란	먹고	바나나	사과	싫은	저는	좋아요
문서1	0	0	0	1	0	1	1	0	0
문서2	0	0	0	1	1	0	1	0	0
문서3	0	1	1	0	2	0	0	0	0
문서4	1	0	0	0	0	0	0	1	1

- 위와 같은 DTM을 Python으로 만들면 다음과 같다.

```
1 import numpy as np  
2  
3 A=np.array([[0,0,0,1,0,1,1,0,0],[0,0,0,1,1,0,1,0,0],[0,1,1,0,2,0,0,0,0],[1,0,0,0,0,0,0,1,1]])  
4 np.shape(A)
```

(4, 9)

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- $4 \times 9$ 의 크기를 가지는 DTM이 생성되었다.
- 이에 대한 풀 SVD(full SVD)를 수행한다.
- 단, 여기서는 대각 행렬의 변수명을  $\Sigma$ 가 아니라  $S$ 를 사용한다. 또한  $V$ 의 전치 행렬을  $VT$ 라고 표현한다.

```
1 U, s, VT = np.linalg.svd(A, full_matrices = True)
```

```
1 print(U.round(2))
2 np.shape(U)
```

```
[[ -0.24  0.75  0.   -0.62]
 [-0.51  0.44 -0.    0.74]
 [-0.83 -0.49 -0.   -0.27]
 [ -0.   -0.    1.    0.   ]]
```

(4, 4)

- Numpy의 **linalg.svd()**는 특이값 분해의 결과로 대각 행렬이 아니라 특이값의 리스트를 반환한다.

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- $4 \times 4$ 의 크기를 가지는 직교 행렬  $U$ 가 생성되었다.
- 대각 행렬  $S$ 를 확인해보자.

```
1 print(s.round(2))  
2 np.shape(s)
```

[2.69 2.05 1.73 0.77]

(4, )

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 특이값을 s에 저장하고 대각 행렬 크기의 행렬을 생성한 후에 그 행렬에 특이값을 삽입하도록 한다.

```
1 S = np.zeros((4, 9)) # 대각 행렬의 크기인 4 x 9의 임의의 행렬 생성
2 S[:4, :4] = np.diag(s) # 특이값을 대각행렬에 삽입
3
4 print(S.round(2))
5 np.shape(S)
```

```
[[2.69 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 2.05 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1.73 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0.77 0. 0. 0. 0. 0.]]
```

(4, 9)

- $4 \times 9$ 의 크기를 가지는 대각 행렬 S가 생성되었다.
- $2.69 > 2.05 > 1.73 > 0.77$  순으로 값이 내림차순을 보이는 것을 확인할 수 있다.

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

```
1 print(VT.round(2))  
2 np.shape(VT)
```

```
[[ -0.   -0.31  -0.31  -0.28  -0.8   -0.09  -0.28  -0.   -0.   ]  
 [ 0.    -0.24  -0.24   0.58  -0.26   0.37   0.58  -0.   -0.   ]  
 [ 0.58  -0.   0.    0.    -0.   0.    -0.   0.58  0.58 ]  
 [ 0.    -0.35  -0.35   0.16   0.25  -0.8   0.16  -0.   -0.   ]  
 [-0.   -0.78  -0.01  -0.2   0.4   0.4   -0.2   0.   0.   ]  
 [-0.29  0.31  -0.78  -0.24   0.23   0.23   0.01  0.14  0.14 ]  
 [-0.29  -0.1   0.26  -0.59  -0.08  -0.08   0.66  0.14  0.14 ]  
 [-0.5   -0.06  0.15   0.24  -0.05  -0.05  -0.19  0.75  -0.25 ]  
 [-0.5   -0.06  0.15   0.24  -0.05  -0.05  -0.19  -0.25  0.75 ]]
```

(9, 9)

- $9 \times 9$ 의 크기를 가지는 직교 행렬 VT(V의 전치 행렬)가 생성되었다.

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 이제  $U \times S \times VT$ 를 하면 기존의 행렬 A가 나와야 한다.
- Numpy의 allclose()는 2개의 행렬이 동일하면 True를 리턴 한다.
- 이를 사용하여 정말로 기존의 행렬 A와 동일한지 확인해보자.

```
1 np.allclose(A, np.dot(np.dot(U,S), VT).round(2))
```

True

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 지금까지 수행한 것은 풀 SVD(Full SVD)이다.
- 이제는 t를 정하고, 절단된 SVD(Truncated SVD)를 수행한다.
- 여기서는  $t = 2$ 로 정한다.
- 우선 대각 행렬  $S$  내의 특이값 중에서 상위 2개만 남기고 제거해보도록 하겠다.

```
1 | S=S[:2,:2]
2 | print(S.round(2))
```

```
[[2.69 0.    ]
 [0.    2.05]]
```

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 상위 2개의 값만 남기고 나머지는 모두 제거된 것을 확인했다.
- 이제 직교 행렬  $U$ 에 대해서도 2개의 열만 남기고 제거한다.

```
1 U = U[:, :2]
2 print(U.round(2))
```

```
[[ -0.24   0.75]
 [-0.51   0.44]
 [-0.83  -0.49]
 [-0.     -0.    ]]
```

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 다음은 행렬 V의 전치 행렬인 VT에 대해서 2개의 행만 남기고 제거한다.
- 이는 VT관점에서는 2개의 열만 남기고 제거한 것이 된다.

```
1 VT = VT[:2, :]  
2 print(VT.round(2))
```

```
[[ -0.       -0.31    -0.31   -0.28   -0.8     -0.09   -0.28   -0.       -0.       ]  
 [ 0.       -0.24    -0.24    0.58   -0.26    0.37    0.58   -0.       -0.       ]]
```

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 이제는 축소된 행렬  $U, S, VT$ 에 대해서 다시  $U \times S \times VT$  연산을 하면 기존의  $A$ 와는 다른 결과가 나오게 된다.
- 값이 손실되었기 때문에 이 세 개의 행렬로는 이제 기존의  $A$  행렬을 복구할 수 없다.
- $U \times S \times VT$  연산을 해서 나오는 값을  $A_{\text{prime}}$ 이라 하고 기존의 행렬  $A$ 와 값을 비교해 본다.

```
1 A_prime = np.dot(np.dot(U,S), VT)
2 print(A)
3 print(A_prime.round(2))
```

```
[[0 0 0 1 0 1 1 0 0]
 [0 0 0 1 1 0 1 0 0]
 [0 1 1 0 2 0 0 0 0]
 [1 0 0 0 0 0 0 1 1]]
 [[ 0.   -0.17  -0.17   1.08   0.12   0.62   1.08  -0.   -0. ]
 [ 0.     0.2    0.2    0.91   0.86   0.45   0.91   0.   0. ]
 [ 0.     0.93   0.93   0.03   2.05  -0.17   0.03   0.   0. ]
 [ 0.     0.     0.     0.     0.    -0.     0.     0.   0. ]]
```

# 잠재 의미 분석(Latent Semantic Analysis, LSA)

- 이제 이렇게 차원이 축소된 U, S, VT의 크기가 어떤 의미를 가지고 있는가?
  - 축소된 U는  $4 \times 2$ 의 크기를 가지는데, 이것은 문서의 개수  $\times$  Topic의 수 t의 크기이다.
  - 단어의 개수인 9는 유지되지 않는데 문서의 개수인 4의 크기가 유지되었으니 4개의 문서 각각을 2개의 값으로 표현하고 있다.
  - 즉, U의 각 행은 잠재 의미를 표현하기 위한 수치화 된 각각의 문서 벡터라고 볼 수 있다.
  - 축소된 VT는  $2 \times 9$ 의 크기를 가지는데, 이것은 Topic의 수 t  $\times$  단어의 개수의 크기이다.
  - VT의 각 열은 잠재 의미를 표현하기 위해 수치화된 각각의 단어 벡터라고 볼 수 있다.

# 실습을 통한 이해

## 1. 준비

- Scikit-learn에서는 Twenty Newsgroups이라고 불리는 20개의 다른 주제를 가진 뉴스 데이터를 제공한다.
- LSA를 사용해서 문서의 수를 원하는 Topic의 수로 압축한 뒤에 각 토픽당 가장 중요한 단어 5개를 출력하는 실습으로 Topic Modeling을 수행해본다.

# 실습을 통한 이해 (Cont.)

## 2. 뉴스 데이터에 대한 이해

```
1 import pandas as pd  
2 from sklearn.datasets import fetch_20newsgroups  
3  
4 dataset = fetch_20newsgroups(shuffle=True, random_state=1, remove=('headers', 'footers', 'quotes'))  
5 documents = dataset.data  
6 len(documents)
```

11314

- 훈련에 사용할 뉴스는 총 11,314개이다.

# 실습을 통한 이해 (Cont.)

## 2. 뉴스 데이터에 대한 이해

- 이 중 첫 번째 훈련용 뉴스를 출력해 본다.

```
1 documents[1]
```

```
"#n#n#n#n#n#nYeah, do you expect people to read the FAQ, etc. and actually accept hard#natheism? No, you need a little leap of  
faith, Jimmy. Your logic runs out#nof steam!#n#n#n#n#n#n#nJim,#n#nSorry I can't pity you, Jim. And I'm sorry that you have the  
se feelings of#ndenial about the faith you need to get by. Oh well, just pretend that it will#nall end happily ever after anyway.  
Maybe if you start a new newsgroup,#nalt.atheist.hard, you won't be bummin' so much?#n#n#n#n#n#n#nBye-Bye, Big Jim. Don't forget  
your Flintstone's Chewables! :)) #n--#nBake Timmons, III"
```

# 실습을 통한 이해 (Cont.)

## 2. 뉴스 데이터에 대한 이해

- 뉴스 데이터에는 특수문자가 포함된 다수의 영어문장으로 구성되어져 있다.
- 이런 형식의 뉴스가 총 11,314개 존재한다.
- Scikit-learn이 제공하는 뉴스 데이터에서 target\_name에는 본래 이 뉴스 데이터가 어떤 20개의 카테고리를 갖고 있었는지 저장되어져 있다.

```
1 print(dataset.target_names)
```

```
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x',  
'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med',  
'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc', 'talk.religion.misc']
```

# 실습을 통한 이해 (Cont.)

## 3. Text Preprocessing

- Text Data에 대해 가능한 정제 과정을 거쳐야만 한다.
- 우선 알파벳을 제외한 구두점, 숫자, 특수 문자를 제거한다.
- 짧은 단어는 유용한 정보를 담고 있지 않다고 가정하고, 길이가 짧은 단어도 제거한다.
- 마지막으로 모든 알파벳을 소문자로 바꿔서 단어의 개수를 줄이는 작업을 한다.

```
1 news_df = pd.DataFrame({'document':documents})
2
3 # 특수 문자 제거
4 news_df['clean_doc'] = news_df['document'].str.replace("[^a-zA-Z]", " ")
5
6 # 길이가 3이하인 단어는 제거 (길이가 짧은 단어 제거)
7 news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: ' '.join([w for w in x.split() if len(w)>3]))
8
9 # 전체 단어에 대한 소문자 변환
10 news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: x.lower())
```

# 실습을 통한 이해 (Cont.)

## 3. Text Preprocessing

- 데이터의 정제가 끝났다.

```
1 news_df['clean_doc'][1]
```

```
'yeah expect people read actually accept hard atheism need little leap faith jimmy your logic runs steam sorry pity sorry that have these feelings denial about faith need well just pretend that will happily ever after anyway maybe start newsgroup atheist hard bummin much forget your flintstone chewables bake timmons'
```

- 뉴스 데이터에서 불용어를 제거한다.
- 불용어를 제거하기 위해서 토큰화를 우선 수행한다.

```
1 from nltk.corpus import stopwords  
2  
3 stop_words = stopwords.words('english')  
4 tokenized_doc = news_df['clean_doc'].apply(lambda x: x.split()) # 토큰화  
5 tokenized_doc = tokenized_doc.apply(lambda x: [item for item in x if item not in stop_words])
```

# 실습을 통한 이해 (Cont.)

## 3. Text Preprocessing

- 확인 차원에서 다시 첫 번째 훈련용 뉴스를 출력한다.

```
1 print(tokenized_doc[1])
```

```
['yeah', 'expect', 'people', 'read', 'actually', 'accept', 'hard', 'atheism', 'need', 'little', 'leap', 'faith', 'jimmy', 'logic',
'runs', 'steam', 'sorry', 'pity', 'sorry', 'feelings', 'denial', 'faith', 'need', 'well', 'pretend', 'happily', 'ever', 'anyway',
'maybe', 'start', 'newsgroup', 'atheist', 'hard', 'bummin', 'much', 'forget', 'flintstone', 'chewables', 'bake', 'timmons']
```

# 실습을 통한 이해 (Cont.)

## 4. TF-IDF 행렬 만들기

- 불용어 제거를 위해 토큰화 작업을 수행하였지만, **TfidfVectorizer**는 기본적으로 토큰화가 되어있지 않은 Text Data를 입력으로 사용한다.
- 따라서 **TfidfVectorizer**를 사용해서 TF-IDF 행렬을 만들기 위해서 다시 토큰화 작업을 역으로 취소하는 작업을 수행해야 한다.
- 이것을 역토큰화(**Detokenization**)라고 한다.

# 실습을 통한 이해 (Cont.)

## 4. TF-IDF 행렬 만들기

```
1 # 역토큰화 (토큰화 작업을 역으로 되돌림)
2 detokenized_doc = []
3
4 for i in range(len(news_df)):
5     t = ' '.join(tokenized_doc[i])
6     detokenized_doc.append(t)
7
8 news_df['clean_doc'] = detokenized_doc
```

- 역토큰화가 제대로 되었는지 다시 첫 번째 훈련용 뉴스를 출력하여 확인해본다.

```
1 news_df['clean_doc'][1]
```

```
'yeah expect people read actually accept hard atheism need little leap faith jimmy logic runs steam sorry pity sorry feelings deny al faith need well pretend happily ever anyway maybe start newsgroup atheist hard bummin much forget flintstone chewables bake tim mons'
```

# 실습을 통한 이해 (Cont.)

## 4. TF-IDF 행렬 만들기

- Scikit-learn의 **TfidfVectorizer**를 통해 단어 1,000개에 대한 TF-IDF 행렬을 만든다.
- Text Data에 있는 모든 단어를 가지고 행렬을 만들 수는 있겠지만, 여기서는 1,000개의 단어로 제한하도록 하겠다.

```
1 from sklearn.feature_extraction.text import TfidfVectorizer  
2  
3 vectorizer = TfidfVectorizer(stop_words='english',  
4 max_features=1000, # 상위 1,000개의 단어를 보존  
5 max_df = 0.5,  
6 smooth_idf=True)  
7  
8 X = vectorizer.fit_transform(news_df['clean_doc'])  
9 X.shape # TF-IDF 행렬의 크기 확인
```

(11314, 1000)

# 실습을 통한 이해 (Cont.)

## 5. Topic Modeling

- Scikit-learn의 절단된 SVD(Truncated SVD)를 사용한다.
- 절단된 SVD를 사용하면 차원을 축소할 수 있다.
- 원래 기존 뉴스 데이터가 20개의 뉴스 카테고리를 갖고 있었기 때문에, 20개의 토픽을 가졌다고 가정한다.
- Topic의 숫자는 **n\_components**의 parameter로 지정이 가능하다.

```
1 from sklearn.decomposition import TruncatedSVD  
2  
3 svd_model = TruncatedSVD(n_components=20, algorithm='randomized', n_iter=100, random_state=122)  
4 svd_model.fit(X)  
5 len(svd_model.components_)
```

# 실습을 통한 이해 (Cont.)

## 5. Topic Modeling

- 여기서 `svd_model.components_`는 앞서 배운 LSA에서 VT에 해당된다.

```
1 np.shape(svd_model.components_)
```

(20, 1000)

- 정확하게 토픽의 수  $t \times$  단어의 수의 크기를 가지는 것을 볼 수 있다.
- 마지막으로 각 20개의 행의 각 1,000개의 열 중 가장 값이 큰 5 개의 값을 찾아서 단어로 출력해보자.

# 실습을 통한 이해 (Cont.)

## 5. Topic Modeling

```
1 terms = vectorizer.get_feature_names() # 단어 집합. 1,000개의 단어가 저장됨.  
2  
3 def get_topics(components, feature_names, n=5):  
4     for idx, topic in enumerate(components):  
5         print("Topic %d:" % (idx+1), [(feature_names[i], topic[i].round(5)) for i in topic.argsort()[-n - 1:-1]])  
6  
7 get_topics(svd_model.components_, terms)
```

Topic 1: [('like', 0.21386), ('know', 0.20046), ('people', 0.19293), ('think', 0.17805), ('good', 0.15128)]  
Topic 2: [('thanks', 0.32888), ('windows', 0.29088), ('card', 0.18069), ('drive', 0.17455), ('mail', 0.15111)]  
Topic 3: [('game', 0.37064), ('team', 0.32443), ('year', 0.28154), ('games', 0.2537), ('season', 0.18419)]  
Topic 4: [('drive', 0.53324), ('scsi', 0.20165), ('hard', 0.15628), ('disk', 0.15578), ('card', 0.13994)]  
Topic 5: [('windows', 0.40399), ('file', 0.25436), ('window', 0.18044), ('files', 0.16078), ('program', 0.13894)]  
Topic 6: [('chip', 0.16114), ('government', 0.16009), ('mail', 0.15625), ('space', 0.1507), ('information', 0.13562)]  
Topic 7: [('like', 0.67086), ('bike', 0.14236), ('chip', 0.11169), ('know', 0.11139), ('sounds', 0.10371)]  
Topic 8: [('card', 0.46633), ('video', 0.22137), ('sale', 0.21266), ('monitor', 0.15463), ('offer', 0.14643)]  
Topic 9: [('know', 0.46047), ('card', 0.33605), ('chip', 0.17558), ('government', 0.1522), ('video', 0.14356)]  
Topic 10: [('good', 0.42756), ('know', 0.23039), ('time', 0.1882), ('bike', 0.11406), ('jesus', 0.09027)]  
Topic 11: [('think', 0.78469), ('chip', 0.10899), ('good', 0.10635), ('thanks', 0.09123), ('clipper', 0.07946)]  
Topic 12: [('thanks', 0.36824), ('good', 0.22729), ('right', 0.21559), ('bike', 0.21037), ('problem', 0.20894)]  
Topic 13: [('good', 0.36212), ('people', 0.33985), ('windows', 0.28385), ('know', 0.26232), ('file', 0.18422)]  
Topic 14: [('space', 0.39946), ('think', 0.23258), ('know', 0.18074), ('nasa', 0.15174), ('problem', 0.12957)]  
Topic 15: [('space', 0.31613), ('good', 0.3094), ('card', 0.22603), ('people', 0.17476), ('time', 0.14496)]  
Topic 16: [('people', 0.48156), ('problem', 0.19961), ('window', 0.15281), ('time', 0.14664), ('game', 0.12871)]  
Topic 17: [('time', 0.34465), ('bike', 0.27303), ('right', 0.25557), ('windows', 0.1997), ('file', 0.19118)]  
Topic 18: [('time', 0.5973), ('problem', 0.15504), ('file', 0.14956), ('think', 0.12847), ('israel', 0.10903)]  
Topic 19: [('file', 0.44163), ('need', 0.26633), ('card', 0.18388), ('files', 0.17453), ('right', 0.15448)]  
Topic 20: [('problem', 0.33006), ('file', 0.27651), ('thanks', 0.23578), ('used', 0.19206), ('space', 0.13185)]

# LSA의 장단점(Pros and Cons of LSA)

## ■ 장점 :

- LSA는 쉽고 빠르게 구현이 가능
- 단어의 잠재적인 의미를 이끌어낼 수 있어 문서의 유사도 계산 등에서 좋은 성능을 보여준다.

## ■ 단점

- SVD의 특성상 이미 계산된 LSA에 새로운 데이터를 추가하여 계산 하려고 하면 보통 처음부터 다시 계산해야 한다.
- 즉, 새로운 정보에 대해 업데이트가 어렵다.

## ■ 대책

- LSA 대신 Word2Vec 등 단어의 의미를 벡터화할 수 있는 또 다른 방법론인 인공 신경망 기반의 방법론을 사용한다.



---

# Topic Modeling

## LDA(Latent Dirichlet Analysis)



# 잠재 디리클레 할당(Latent Dirichlet Allocation, LDA)

- Topic Modeling은 문서의 집합에서 Topic을 찾아내는 Process이다.
- 이것으로 검색 엔진, 고객 민원 시스템 등과 같이 문서의 주제를 알아내는 일이 중요한 곳에서 사용된다.
- 잠재 디리클레 할당(Latent Dirichlet Allocation, LDA)은 Topic Modeling의 대표적인 알고리즘이다.
- LDA는 문서들은 Topic들의 혼합으로 구성되어져 있으며, Topic들은 확률 분포에 기반하여 단어들을 생성한다고 가정한다.
- 데이터가 주어지면, LDA는 문서가 생성되던 과정을 역추적 한다.

# 잠재 디리클레 할당(Latent Dirichlet Allocation, LDA) (Cont.)

- 우선 LDA의 내부 메커니즘에 대해서 이해하기 전에, LDA를 일종의 블랙 박스로 보고 LDA에 문서 집합을 입력하면, 어떤 결과를 보여주는지 간소화 된 예를 보자.
- 아래와 같은 3개의 문서가 있다고 가정한다.
  - 문서1 : 저는 사과랑 바나나를 먹어요
  - 문서2 : 우리는 귀여운 강아지가 좋아요
  - 문서3 : 저의 깜찍하고 귀여운 강아지가 바나나를 먹어요

# 잠재 디리클레 할당(Latent Dirichlet Allocation, LDA) (Cont.)

- LDA는 LDA를 수행할 때 문서 집합에서 Topic이 몇 개가 존재할지 가정하는 것은 사용자가 해야 할 일이다.
- 여기서는 LDA에 2개의 토픽을 찾으라고 요청하겠다.
- 즉, Topic의 개수를 의미하는 변수를  $k$ 라고 하였을 때,  $k$ 를 2로 한다는 의미이다.
- $k$ 의 값을 잘못 선택하면 원치 않는 이상한 결과가 나올 수 있다.
- 이것을 Hyper-Parameter라고 한다.

# 잠재 디리클레 할당(Latent Dirichlet Allocation, LDA) (Cont.)

- LDA가 앞의 세 문서로부터 2개의 토픽을 찾은 결과는 아래와 같다.
- 여기서는 LDA 입력 전에 주어와 불필요한 조사 등을 제거하는 전처리 과정은 거쳤다고 가정한다.
- 즉, 전처리 과정을 거친 DTM이 LDA의 입력이 되었다고 가정한다.

# 잠재 디리클레 할당(Latent Dirichlet Allocation, LDA) (Cont.)

- LDA는 각 문서의 Topic 분포와 각 Topic 내의 단어 분포를 추정한다.
- 각 문서의 Topic 분포
  - 문서 1 : Topic A 100%
  - 문서 2 : Topic B 100%
  - 문서 3 : Topic B 60%, Topic A 40%
- 각 Topic의 단어 분포
  - Topic A : 사과 20%, 바나나 40%, 먹어요 40%, 귀여운 0%, 강아지 0%, 깜찍하고 0%, 좋아요 0%
  - Topic B : 사과 0%, 바나나 0%, 먹어요 0%, 귀여운 33%, 강아지 33%, 깜직하고 16%, 좋아요 16%
- LDA는 Topic의 제목을 정해주지 않지만, 이 시점에서 알고리즘의 사용자는 위 결과로부터 두 Topic이 각각 과일에 대한 토픽과 강아지에 대한 토픽이라고 판단해볼 수 있다.

# LDA의 가정

- LDA는 문서의 집합으로부터 어떤 Topic이 존재하는지를 알아내기 위한 Algorithm이다.
- LDA는 앞서 배운 빈도수 기반의 표현 방법인 BoW의 행렬 DTM 또는 TF-IDF 행렬을 입력으로 하는데, 이로부터 알 수 있는 사실은 LDA는 단어의 순서는 신경 쓰지 않겠다는 전제가 있다.
- LDA는 문서들로부터 Topic을 뽑아내기 위해서 이러한 가정을 전제로 두고 있다.

## LDA의 가정 (Cont.)

■ 각각의 문서는 다음과 같은 과정을 거쳐서 작성되었다고 가정한다.

1. 문서에 사용할 단어의 개수 N을 정한다.
  - 예) 5개의 단어를 정하였다.
2. 문서에 사용할 Topic의 혼합을 확률 분포에 기반하여 결정한다.
  - 예) 위 예제와 같이 Topic이 2개라고 하였을 때 강아지 토픽을 60%, 과일 토픽을 40%와 같이 선택할 수 있다.

# LDA의 가정 (Cont.)

## 3. 문서에 사용할 각 단어를 다음과 같이 정한다.

① Topic 분포에서 Topic T를 확률적으로 고른다.

- 예) 60% 확률로 강아지 Topic을 선택하고, 40% 확률로 과일 Topic을 선택 할 수 있다.

② 선택한 Topic T에서 단어의 출현 확률 분포에 기반해 문서에 사용할 단어를 고른다.

- 예) 강아지 Topic을 선택하였다면, 33% 확률로 강아지란 단어를 선택할 수 있다.
- 이제 3.을 반복하면서 문서를 완성한다.

■ 이러한 과정을 통해 문서가 작성되었다는 가정 하에 LDA는 토픽을 뽑아내기 위하여 위 과정을 역으로 추적하는 역공학 (reverse engineering)을 수행하는 것이다.

# LDA 수행하기

■ LDA의 수행 과정을 정리해 보자.

1. 사용자는 Algorithm에게 Topic의 개수  $k$ 를 알려준다.

- LDA는 토픽의 개수  $k$ 를 입력 받으면,  $k$ 개의 Topic이  $M$ 개의 전체 문서에 걸쳐 분포되어 있다고 가정한다.

2. 모든 단어를  $k$ 개 중 하나의 Topic에 할당한다.

- 이제 LDA는 모든 문서의 모든 단어에 대해서  $k$ 개 중 하나의 Topic을 random으로 할당한다.
- 이 작업이 끝나면 각 문서는 Topic을 가지며, Topic은 단어 분포를 가지는 상태이다.
- 물론 random로 할당하였기 때문에 사실 이 결과는 전부 틀린 상태이다.
- 만약 한 단어가 한 문서에서 2회 이상 등장했다면, 각 단어는 서로 다른 Topic에 할당되었을 수도 있다.

# LDA 수행하기 (Cont.)

■ LDA의 수행 과정을 정리해 보자.

3. 이제 모든 문서의 모든 단어에 대해서 아래의 사항을 반복 진행한다. (iterative)

- 어떤 문서의 각 단어  $w$ 는 자신은 잘못된 Topic에 할당되어져 있지만, 다른 단어들은 전부 올바른 Topic에 할당되어져 있는 상태라고 가정한다.
- 이에 따라 단어  $w$ 는 아래의 두 가지 기준에 따라서 Topic이 재할당된다.
  - $p(\text{topic } t \mid \text{document } d)$  : 문서  $d$ 의 단어들 중 Topic  $t$ 에 해당하는 단어들의 비율
  - $p(\text{word } w \mid \text{topic } t)$  : 단어  $w$ 를 갖고 있는 모든 문서들 중 Topic  $t$ 가 할당된 비율
- 이를 반복하면, 모든 할당이 완료된 수렴 상태가 된다.

# LDA 수행하기 (Cont.)

doc1

word	apple	banana	apple	dog	dog
topic	B	B	???	A	A

doc2

word	cute	book	king	apple	apple
topic	B	B	B	B	B

- 위의 그림은 두 개의 문서 doc1과 doc2를 보여준다.
- 여기서는 doc1의 세 번째 단어 apple의 Topic을 결정하려고 한다.

# LDA 수행하기 (Cont.)

- 첫 번째로 사용하는 기준은 문서 doc1의 단어들이 어떤 Topic에 해당하는지를 보는 것이다.
- doc1의 모든 단어들은 토픽 A와 토픽 B에 50 대 50의 비율로 할당되어져 있으므로, 이 기준에 따르면 단어 apple은 토픽 A 또는 토픽 B 둘 중 어디에도 속할 가능성이 있는 것이다.

doc1

word	apple	banana	apple	dog	dog
topic	B	B	???	A	A

doc2

word	cute	book	king	apple	apple
topic	B	B	B	B	B

# LDA 수행하기 (Cont.)

- 두 번째 기준은 단어 apple이 전체 문서에서 어떤 토픽에 할당되어져 있는지를 보는 것이다.
- 이 기준에 따르면 단어 apple은 토픽 B에 할당될 가능성이 높다.
- 이러한 두 가지 기준을 참고하여 LDA는 doc1의 apple을 어떤 토픽에 할당할지 결정한다.

doc1

word	apple	banana	apple	dog	dog
topic	B	B	???	A	A

doc2

word	cute	book	king	apple	apple
topic	B	B	B	B	B

# LDA와 LSA의 차이

## ■ LSA

- DTM을 차원 축소 하여 축소 차원에서 근접 단어들을 Topic으로 묶는다.

## ■ LDA

- 단어가 특정 Topic에 존재할 확률과 문서에 특정 Topic이 존재할 확률을 결합확률로 추정하여 Topic을 추출한다.

# 실습을 통한 이해

## 1. 정수 Encoding과 단어 집합 만들기

- 바로 이전 LSA 챕터에서 사용하였던 Twenty Newsgroups이라고 불리는 20개의 다른 주제를 가진 뉴스 데이터를 다시 사용하기로 한다.
- 전처리 과정은 이전 실습과 중복되므로 생략한다.
- 동일한 전처리 과정을 거친 후에 tokenized\_doc으로 저장한 상태라고 가정한다.
- 훈련용 뉴스를 5개만 출력해서 확인한다.

# 실습을 통한 이해 (Cont.)

## 1. 정수 Encoding과 단어 집합 만들기

```
1 import pandas as pd
2 from sklearn.datasets import fetch_20newsgroups
3
4 dataset = fetch_20newsgroups(shuffle=True, random_state=1, remove=('headers', 'footers', 'quotes'))
5 documents = dataset.data
6 len(documents)
```

11314

```
1 documents[1]
```

"Yeah, do you expect people to read the FAQ, etc. and actually accept hardatheism? No, you need a little leap of faith, Jimmy. Your logic runs out of steam! Jim Sorry I can't pity you, Jim. And I'm sorry that you have these feelings of denial about the faith you need to get by. Oh well, just pretend that it will all end happily ever after anyway. Maybe if you start a new newsgroup, alt.atheist.hard, you won't be bummin' so much? Bye-Bye, Big Jim. Don't forget your Flintstone's Chewables! :) --Bake Timmons, III"

```
1 print(dataset.target_names)
```

['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc', 'talk.religion.misc']

# 실습을 통한 이해(Cont.)

## 1. 정수 Encoding과 단어 집합 만들기

```
1 news_df = pd.DataFrame({'document':documents})  
2  
3 # 특수 문자 제거  
4 news_df['clean_doc'] = news_df['document'].str.replace("[^a-zA-Z]", " ")  
5  
6 # 길이가 3이하인 단어는 제거 (길이가 짧은 단어 제거)  
7 news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: ' '.join([w for w in x.split() if len(w)>3]))  
8  
9 # 전체 단어에 대한 소문자 변환  
10 news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: x.lower())
```

```
1 news_df['clean_doc'][1]
```

'yeah expect people read actually accept hard atheism need little leap faith jimmy your logic runs steam sorry pity sorry that have these feelings denial about faith need well just pretend that will happily ever after anyway maybe start newsgroup atheist hard bummin much forget your flintstone chewables bake timmons'

```
1 from nltk.corpus import stopwords  
2  
3 stop_words = stopwords.words('english')  
4 tokenized_doc = news_df['clean_doc'].apply(lambda x: x.split()) # 토큰화  
5 tokenized_doc = tokenized_doc.apply(lambda x: [item for item in x if item not in stop_words])
```

```
1 print(tokenized_doc[:5])
```

```
0 [well, sure, story, seem, biased, disagree, st...  
1 [yeah, expect, people, read, actually, accept,...  
2 [although, realize, principle, strongest, poin...  
3 [notwithstanding, legitimate, fuss, proposal, ...  
4 [well, change, scoring, playoff, pool, unfortu...  
Name: clean_doc, dtype: object
```

# 실습을 통한 이해(Cont.)

## 1. 정수 Encoding과 단어 집합 만들기

- 각 단어에 정수 Encoding을 하는 동시에, 각 뉴스에서의 단어의 빈도수를 기록한다.
- 여기서는 각 단어를 (**word\_id**, **word\_frequency**)의 형태로 바꾸려고 한다.
- **word\_id**는 단어가 정수 Encoding된 값이고, **word\_frequency**는 해당 뉴스에서의 해당 단어의 빈도수를 의미한다.
- 이는 **gensim**의 **corpora.Dictionary()**를 사용하여 손쉽게 구할 수 있다.
- 전체 뉴스에 대해서 정수 Encoding을 수행하고, 두 번째 뉴스를 출력한다.

# 실습을 통한 이해(Cont.)

## 1. 정수 Encoding과 단어 집합 만들기

```
1 from gensim import corpora  
2  
3 dictionary = corpora.Dictionary(tokenized_doc)  
4 corpus = [dictionary.doc2bow(text) for text in tokenized_doc]  
5  
6 print(corpus[1]) # 수행된 결과에서 두번째 뉴스 출력. 첫번째 문서의 Index는 0  
  
[(52, 1), (55, 1), (56, 1), (57, 1), (58, 1), (59, 1), (60, 1), (61, 1), (62, 1), (63, 1), (64, 1), (65, 1), (66, 2), (67, 1), (68, 1), (69, 1), (70, 1), (71, 2), (72, 1), (73, 1), (74, 1), (75, 1), (76, 1), (77, 1), (78, 2), (79, 1), (80, 1), (81, 1), (82, 1), (83, 1), (84, 1), (85, 2), (86, 1), (87, 1), (88, 1), (89, 1)]
```

- 위의 출력 결과 중에서 (66, 2)는 정수 Encoding이 66으로 할당된 단어가 두 번째 뉴스에서는 두 번 등장하였음을 의미한다.
- 66이라는 값을 가지는 단어가 정수 Encoding이 되기 전에는 어떤 단어였는지 확인하여 보자. 봅시다. 이는 dictionary[]에 기준 단어가 무엇인지 알고자하는 정수값을 입력하여 확인할 수 있습니다.

# 실습을 통한 이해(Cont.)

## 1. 정수 Encoding과 단어 집합 만들기

- 66이라는 값을 가지는 단어가 정수 Encoding이 되기 전에는 어떤 단어였는지 확인하여 보자.
- 방법은 **dictionary[]**에 기존 단어가 무엇인지 알고자 하는 정수값을 입력하여 확인할 수 있다.

```
1 print(dictionary[66])
```

faith

- 기존에 단어가 faith이었음을 알 수 있다.
- 총 학습된 단어의 개수를 확인해 보자.

```
1 len(dictionary)
```

64281

# 실습을 통한 이해(Cont.)

## 2. LDA 모델 훈련시키기

- 기존의 뉴스 데이터가 총 20개의 카테고리를 가지고 있으므로 Topic의 개수를 20으로 하여 LDA 모델을 학습하게 한다.

```
1 import gensim
2 NUM_TOPICS = 20 #20개의 토픽, k=20
3 ldamodel = gensim.models.ldamodel.LdaModel(corpus, num_topics = NUM_TOPICS, id2word=dictionary, passes=15)
4 topics = ldamodel.print_topics(num_words=4)
5 for topic in topics:
6     print(topic)
```

- 맨 앞에 있는 토픽 번호는 0부터 시작하므로 총 20개의 토픽은 0부터 19까지의 번호가 할당되어져 있다.
- passes**는 알고리즘의 동작 횟수를 말하는데, 알고리즘이 결정하는 토픽의 값이 적절히 수렴할 수 있도록 충분히 적당한 횟수를 정해주면 된다.
- 여기서는 총 15회를 수행하였다.
- 여기서는 **num\_words**=4로 총 4개의 단어만 출력하도록 하였다.

# 실습을 통한 이해(Cont.)

## 2. LDA 모델 훈련시키기

(0, '0.018\*"armenian" + 0.016\*"armenians" + 0.013\*"turkish" + 0.008\*"greek"')  
(1, '0.015\*"montreal" + 0.011\*"lost" + 0.011\*"john" + 0.010\*"braves"')  
(2, '0.020\*"jesus" + 0.012\*"christian" + 0.011\*"bible" + 0.011\*"believe"')  
(3, '0.014\*"would" + 0.011\*"people" + 0.007\*"think" + 0.007\*"many"')  
(4, '0.035\*"stream" + 0.026\*"contest" + 0.012\*"null" + 0.010\*"icon"')  
(5, '0.018\*"nist" + 0.014\*"ncsl" + 0.013\*"sabbath" + 0.011\*"maine"')  
(6, '0.023\*"game" + 0.020\*"team" + 0.015\*"year" + 0.015\*"games"')  
(7, '0.019\*"drive" + 0.013\*"card" + 0.011\*"scsi" + 0.010\*"disk"')  
(8, '0.010\*"good" + 0.008\*"like" + 0.006\*"much" + 0.006\*"bike"')  
(9, '0.026\*"tobacco" + 0.016\*"smokeless" + 0.014\*"food" + 0.011\*"fallacy"')  
(10, '0.011\*"windows" + 0.009\*"available" + 0.009\*"file" + 0.009\*"mail"')  
(11, '0.013\*"encryption" + 0.012\*"chip" + 0.011\*"keys" + 0.010\*"security"')  
(12, '0.012\*"president" + 0.009\*"national" + 0.008\*"states" + 0.008\*"health"')  
(13, '0.016\*"gordon" + 0.015\*"pitt" + 0.014\*"soon" + 0.014\*"banks"')  
(14, '0.025\*"image" + 0.021\*"color" + 0.017\*"display" + 0.013\*"screen"')  
(15, '0.029\*"file" + 0.023\*"entry" + 0.021\*"output" + 0.017\*"program"')  
(16, '0.015\*"would" + 0.011\*"know" + 0.011\*"people" + 0.011\*"like"')  
(17, '0.018\*"israel" + 0.016\*"jews" + 0.011\*"israeli" + 0.008\*"jewish"')  
(18, '0.038\*"space" + 0.014\*"nasa" + 0.008\*"launch" + 0.007\*"data"')  
(19, '0.013\*"event" + 0.012\*"water" + 0.008\*"mask" + 0.008\*"chain"')

- 각 단어 앞에 붙은 수치는 단어의 해당 Topic에 대한 기여도를 보여준다.

# 실습을 통한 이해(Cont.)

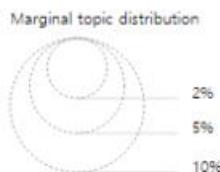
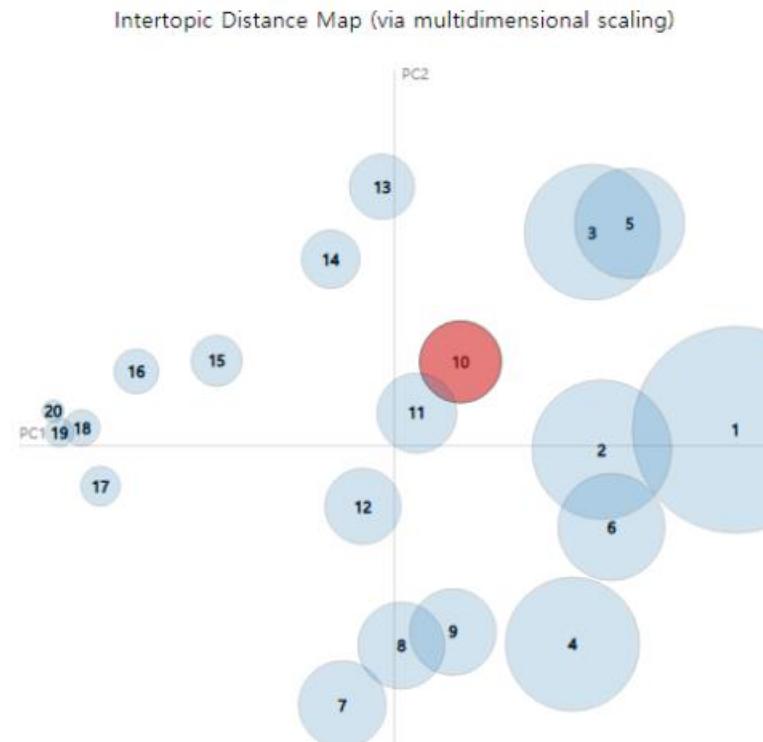
## 3. LDA 시각화 하기

- LDA 시각화를 위해서는 **pyLDAvis**의 설치가 필요하다.
- Terminal에서 아래의 명령을 수행하여 **pyLDAvis**를 설치한다.  
\$ **pip install pyLDAvis**
- 설치가 완료되었다면 LDA 시각화 실습을 진행한다.

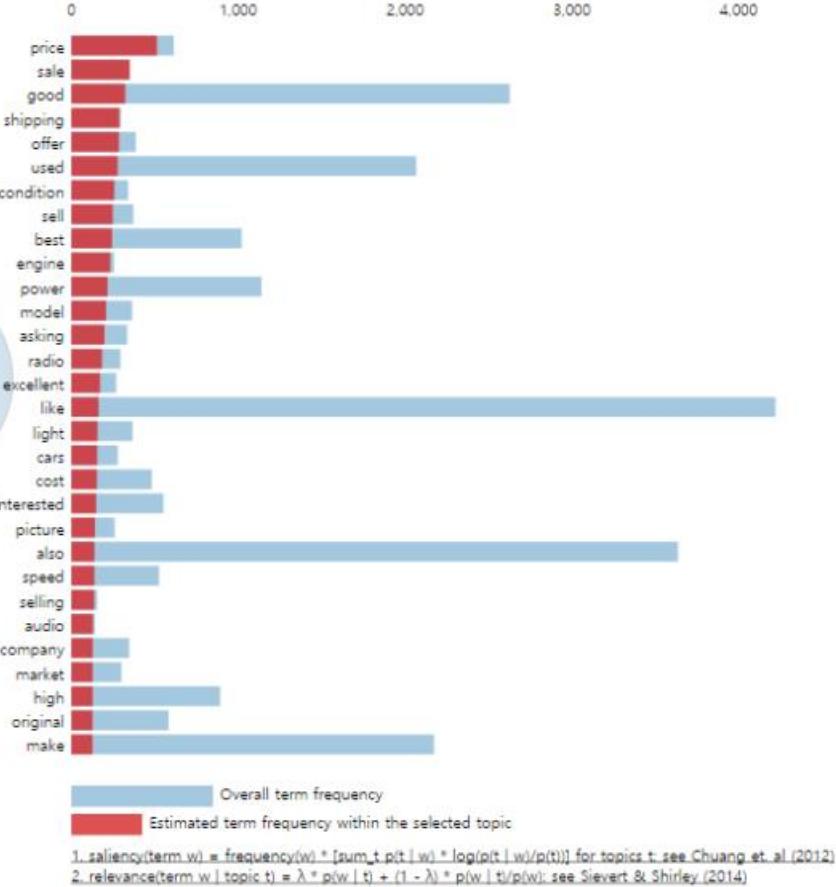
```
1 import pyLDAvis.gensim  
2  
3 pyLDAvis.enable_notebook()  
4 vis = pyLDAvis.gensim.prepare(ldamodel, corpus, dictionary)  
5 pyLDAvis.display(vis)
```

# 실습을 통한 이해(Cont.)

## 3. LDA 시각화 하기



Top-30 Most Relevant Terms for Topic 10 (3.7% of tokens)



# 실습을 통한 이해(Cont.)

## 3. LDA 시각화 하기

- 좌측의 원들은 각각의 20개의 Topic을 나타낸다.
- 각 원과의 거리는 각 Topic들이 서로 얼마나 다른지를 보여준다.
- 만약 두 개의 원이 겹친다면, 이 두 개의 Topic은 유사한 Topic이라는 의미이다.
- 앞의 그림에서는 10번 토픽을 클릭하였고, 이에 따라 우측에는 10번 Topic에 대한 정보가 나타난다.
- 한 가지 주의할 점은 LDA 모델의 출력 결과에서는 토픽 번호가 0부터 할당되어 0~19의 숫자가 사용된 것과는 달리 위의 LDA 시각화에서는 토픽의 번호가 1부터 시작하므로 각 토픽 번호는 이제 +1이 된 값인 1~20까지의 값을 가진다는 것이다.

# 실습을 통한 이해(Cont.)

## 4. 문서 별 Topic 분포 보기

- 앞에서 Topic 별 단어 분포는 확인하였으나, 아직 문서 별 Topic 분포에 대해서는 확인하지 못했다.
- 각 문서의 Topic 분포는 이미 훈련된 LDA 모델인 **ldamodel[]**에 전체 데이터가 정수 Encoding 된 결과를 넣은 후에 확인이 가능하다.
- 상위 5개의 문서에 대해서만 토픽 분포를 확인해보자.

```
1 for i, topic_list in enumerate(ldamodel[corpus]):  
2     if i==5:  
3         break  
4     print(i,'번째 문서의 topic 비율은',topic_list)
```

```
0 번째 문서의 topic 비율은 [(0, 0.081939295), (3, 0.3149616), (17, 0.5893812)]  
1 번째 문서의 topic 비율은 [(1, 0.027499277), (2, 0.28059345), (4, 0.027610902), (13, 0.05387194), (16, 0.5632455), (17, 0.028736576)]  
2 번째 문서의 topic 비율은 [(3, 0.07977432), (16, 0.5703075), (17, 0.3361781)]  
3 번째 문서의 topic 비율은 [(0, 0.03080709), (2, 0.034027234), (3, 0.19099821), (6, 0.050004523), (7, 0.08839382), (11, 0.32400486), (13, 0.01566251), (16, 0.25667217)]  
4 번째 문서의 topic 비율은 [(1, 0.374414), (6, 0.12839727), (15, 0.07737734), (16, 0.39017493)]
```

# 실습을 통한 이해(Cont.)

## 4. 문서 별 Topic 분포 보기

- 앞(아래)의 출력 결과에서 (숫자, 확률)은 각각 Topic 번호와 해당 Topic이 해당 문서에서 차지하는 분포도를 의미한다.
- 예) 0번째 문서의 토픽 비율에서 (0, 0.081939295)은 0번 토픽이 8%의 분포도를 가지는 것을 의미한다.

```
1 for i, topic_list in enumerate(ldamodel[corpus]):  
2     if i==5:  
3         break  
4     print(i,'번째 문서의 topic 비율은',topic_list)
```

```
0 번째 문서의 topic 비율은 [(0, 0.081939295), (3, 0.3149616), (17, 0.5893812)]  
1 번째 문서의 topic 비율은 [(1, 0.027499277), (2, 0.28059345), (4, 0.027610902), (13, 0.05387194), (16, 0.5632455), (17, 0.028736576)]  
2 번째 문서의 topic 비율은 [(3, 0.07977432), (16, 0.5703075), (17, 0.3361781)]  
3 번째 문서의 topic 비율은 [(0, 0.03080709), (2, 0.034027234), (3, 0.19099821), (6, 0.050004523), (7, 0.08839382), (11, 0.32400486), (13, 0.01566251), (16, 0.25667217)]  
4 번째 문서의 topic 비율은 [(1, 0.374414), (6, 0.12839727), (15, 0.07737734), (16, 0.39017493)]
```

# 실습을 통한 이해(Cont.)

## 4. 문서 별 Topic 분포 보기

- 앞의 코드를 응용하여 좀 더 깔끔한 형태인 DataFrame 형식으로 출력한다.

```
1 def make_topictable_per_doc(ldamodel, corpus, texts):
2     topic_table = pd.DataFrame()
3
4     # 몇 번째 문서인지와 의미하는 문서 번호와 해당 문서의 토픽 비중을 한 줄씩 꺼내온다.
5     for i, topic_list in enumerate(ldamodel[corpus]):
6         doc = topic_list[0] if ldamodel.per_word_topics else topic_list
7         doc = sorted(doc, key=lambda x: (x[1]), reverse=True)
8         # 각 문서에 대해서 비중이 높은 토픽순으로 토픽을 정렬한다.
9         # EX) 정렬 전 0번 문서 : (2번 토픽, 48.5%), (8번 토픽, 25%), (10번 토픽, 5%), (12번 토픽, 21.5%),
10        # Ex) 정렬 후 0번 문서 : (2번 토픽, 48.5%), (8번 토픽, 25%), (12번 토픽, 21.5%), (10번 토픽, 5%)
11        # 48 > 25 > 21 > 5 순으로 정렬이 된 것.
12
13     # 모든 문서에 대해서 각각 아래를 수행
14     for j, (topic_num, prop_topic) in enumerate(doc): # 몇 번 토픽인지와 비중을 나눠서 저장한다.
15         if j == 0: # 정렬을 한 상태이므로 가장 앞에 있는 것이 가장 비중이 높은 토픽
16             topic_table = topic_table.append(pd.Series([int(topic_num), round(prop_topic,4), topic_list]), ignore_index=True)
17             # 가장 비중이 높은 토픽과, 가장 비중이 높은 토픽의 비중과, 전체 토픽의 비중을 저장한다.
18         else:
19             break
20
return(topic_table)
```

# 실습을 통한 이해(Cont.)

## 4. 문서 별 Topic 분포 보기

```
1 topictable = make_topictable_per_doc(Idamodel, corpus, tokenized_doc)
2 topictable = topictable.reset_index() # 문서 번호를 의미하는 열(column)로 사용하기 위해서 인덱스 열을 하나 더 만든다.
3 topictable.columns = ['문서 번호', '가장 비중이 높은 토픽', '가장 높은 토픽의 비중', '각 토픽의 비중']
4 topictable[:10]
```

	문서 번호	가장 비중이 높은 토픽	가장 높은 토픽의 비중	각 토픽의 비중
0	0	17.0	0.5894	[(0, 0.08194105), (3, 0.3149543), (17, 0.58938...]
1	1	16.0	0.5633	[(1, 0.027499277), (2, 0.28058016), (4, 0.0276...]
2	2	16.0	0.5704	[(3, 0.07971381), (16, 0.57035494), (17, 0.336...]
3	3	11.0	0.3240	[(0, 0.030799083), (2, 0.034107946), (3, 0.190...]
4	4	16.0	0.3902	[(1, 0.3744112), (6, 0.12840377), (15, 0.07737...]
5	5	2.0	0.6273	[(2, 0.62731075), (13, 0.05384767), (16, 0.280...]
6	6	7.0	0.3044	[(3, 0.26125026), (7, 0.30444682), (8, 0.20588...]
7	7	16.0	0.5930	[(3, 0.21724717), (11, 0.022008713), (12, 0.02...]
8	8	3.0	0.7133	[(3, 0.7132578), (9, 0.062096816), (10, 0.1995...]
9	9	8.0	0.3460	[(3, 0.15711968), (7, 0.08364172), (8, 0.34598...]



# Topic Modeling



## LDA(Latent Dirichlet Analysis)

### Lab

# 뉴스 기사 제목 데이터에 대한 이해

- 여기서 사용할 데이터는 약 15년 동안 발행되었던 뉴스 기사 제목을 모아놓은 영어 데이터이다.
- 해당 데이터는 아래 링크에서 다운받을 수 있다.
- 링크 : <https://www.kaggle.com/therohk/million-headlines>

```
1 import pandas as pd  
2 data = pd.read_csv(r'./abcnews-date-text.csv', error_bad_lines=False)
```

```
1 print(len(data))
```

1103663

## 뉴스 기사 제목 데이터에 대한 이해 (Cont.)

- 해당 데이터는 약 100만개의 샘플을 갖고 있다.
- 상위 5개의 샘플만 출력해 본다.

```
1 print(data.head(5))
```

	publish_date	headline_text
0	20030219	aba decides against community broadcasting lic...
1	20030219	act fire witnesses must be aware of defamation
2	20030219	a g calls for infrastructure protection summit
3	20030219	air nz staff in aust strike for pay rise
4	20030219	air nz strike to affect australian travellers

# 뉴스 기사 제목 데이터에 대한 이해 (Cont.)

- 이 데이터는 `publish_data`와 `headline_text`라는 두 개의 열을 갖고 있다.
- 각각 뉴스가 나온 날짜와 뉴스 기사 제목을 의미한다.
- 필요한 데이터는 이 중에서 `headline_text` 열.
- 즉, 뉴스 기사 제목이므로 이 부분만 별도로 저장한다.

```
1 text = data[['headline_text']]  
2 text.head(5)
```

## headline\_text

0	aba decides against community broadcasting lic...
1	act fire witnesses must be aware of defamation
2	a g calls for infrastructure protection summit
3	air nz staff in aust strike for pay rise
4	air nz strike to affect australian travellers

# 텍스트 전처리

- 토큰화, 불용어 제거, 표제어 추출이라는 전처리를 수행한다.
- NLTK의 word\_tokenize()를 통해 단어 Token화를 수행한다.

```
1 from nltk.tokenize import word_tokenize  
2 text['headline_text'] = text.apply(lambda row: nltk.word_tokenize(row['headline_text']), axis=1)
```

- 상위 5개의 샘플만 추출하여 단어 Token화 결과를 확인한다.

```
1 print(text.head())
```

	headline_text
0	[aba, decides, against, community, broadcastin...]
1	[act, fire, witnesses, must, be, aware, of, de...]
2	[a, g, calls, for, infrastructure, protection,...]
3	[air, nz, staff, in, aust, strike, for, pay, r...]
4	[air, nz, strike, to, affect, australian, trav...]

# 텍스트 전처리 (Cont.)

## ■ 불용어를 제거한다.

```
1 from nltk.corpus import stopwords  
2 stop = stopwords.words('english')  
3 text['headline_text'] = text['headline_text'].apply(lambda x: [word for word in x if word not in (stop)])
```

```
1 print(text.head(5))
```

```
headline_text  
0 [aba, decides, community, broadcasting, licence]  
1 [act, fire, witnesses, must, aware, defamation]  
2 [g, calls, infrastructure, protection, summit]  
3 [air, nz, staff, aust, strike, pay, rise]  
4 [air, nz, strike, affect, australian, travellers]
```

- 상위 5개의 샘플에 대해서 불용어를 제거하기 전과 후의 데이터만 비교해도 확실히 몇 가지 단어들이 사라진 것이 보일 것이다.
- against, be, of, a, in, to 등의 단어가 제거되었다.

# 텍스트 전처리 (Cont.)

- 이제 표제어 추출을 수행한다.
- 표제어 추출로 3인칭 단수 표현을 1인칭으로 바꾸고, 과거 현재형 동사를 현재형으로 바꾼다.

```
1 from nltk.stem import WordNetLemmatizer
2 text['headline_text'] = text['headline_text'].apply(lambda x: [WordNetLemmatizer().lemmatize(word, pos='v') for word in x])
3 print(text.head(5))

headline_text
0    [aba, decide, community, broadcast, licence]
1    [act, fire, witness, must, aware, defamation]
2    [g, call, infrastructure, protection, summit]
3    [air, nz, staff, aust, strike, pay, rise]
4    [air, nz, strike, affect, australian, travellers]
```

# 텍스트 전처리 (Cont.)

- 길이가 3이하인 단어에 대해서 제거하는 작업을 수행한다.
- 이번에는 결과를 **tokenized\_doc**이라는 새로운 변수에 저장한다.

```
1 tokenized_doc = text['headline_text'].apply(lambda x: [word for word in x if len(word) > 3])
2 print(tokenized_doc[:5])
```

```
0      [decide, community, broadcast, licence]
1      [fire, witness, must, aware, defamation]
2      [call, infrastructure, protection, summit]
3          [staff, aust, strike, rise]
4      [strike, affect, australian, travellers]
Name: headline_text, dtype: object
```

# TF-IDF 행렬 만들기

- **TfidfVectorizer**는 기본적으로 token화가 되어있지 않은 Text Data를 입력으로 사용한다.
- **TfidfVectorizer**를 사용해서 TF-IDF 행렬을 만들기 위해서 다시 token화 작업을 역으로 취소하는 역토큰화 (*Detokenization*)작업을 수행한다.

```
1 # 역토큰화 (토큰화 작업을 되돌림)
2 detokenized_doc = []
3 for i in range(len(text)):
4     t = ' '.join(tokenized_doc[i])
5     detokenized_doc.append(t)
6
7 text['headline_text'] = detokenized_doc # 다시 text['headline_text']에 재저장
```

## TF-IDF 행렬 만들기 (Cont.)

- 역토큰화가 되었는지 `text['headline_text']`의 5개의 샘플을 출력해 본다.

```
1 text['headline_text'][:5]
```

```
0      decide community broadcast licence
1      fire witness must aware defamation
2      call infrastructure protection summit
3                      staff aust strike rise
4      strike affect australian travellers
Name: headline_text, dtype: object
```

# TF-IDF 행렬 만들기 (Cont.)

- Scikit-learn의 TfidfVectorizer으로 TF-IDF 행렬을 만든다.
- Text Data에 있는 모든 단어를 가지고 행렬을 만들 수도 있겠지만, 여기서는 간단히 1,000개의 단어로 제한하겠다.

```
1 from sklearn.feature_extraction.text import TfidfVectorizer  
2  
3 vectorizer = TfidfVectorizer(stop_words='english', max_features= 1000) # 상위 1,000개의 단어를 보존  
4 X = vectorizer.fit_transform(text['headline_text'])  
5 X.shape # TF-IDF 행렬의 크기 확인
```

(1103663, 1000)

# Topic Modeling

```
1 from sklearn.decomposition import LatentDirichletAllocation  
2 lda_model=LatentDirichletAllocation(n_components=10,learning_method='online',random_state=777,max_iter=1)
```

```
1 lda_top=lda_model.fit_transform(X)
```

```
1 print(lda_model.components_)  
2 print(lda_model.components_.shape)
```

```
[[1.00000703e-01 1.00000829e-01 1.00003578e-01 ... 1.00004871e-01  
 1.00003129e-01 1.00002930e-01]  
[1.00001421e-01 8.66862951e+02 1.00008903e-01 ... 1.00004224e-01  
 1.00005598e-01 7.01841034e+02]  
[1.00000648e-01 1.00000545e-01 1.00002661e-01 ... 1.00005158e-01  
 1.00008596e-01 1.00001987e-01]  
...  
[1.00001636e-01 1.00000889e-01 2.68570402e+03 ... 1.00003039e-01  
 1.00010511e-01 1.00004475e-01]  
[1.00001352e-01 1.00000852e-01 1.00003353e-01 ... 1.00003378e-01  
 1.00005211e-01 1.00003635e-01]  
[1.00002244e-01 1.00000967e-01 1.00003675e-01 ... 1.00002444e-01  
 1.00003580e-01 1.00004738e-01]]  
(10, 1000)
```

# Topic Modeling (Cont.)

```
1 terms = vectorizer.get_feature_names() # 단어 집합. 1,000개의 단어가 저장됨.  
2  
3 def get_topics(components, feature_names, n=5):  
4     for idx, topic in enumerate(components):  
5         print("Topic %d:" % (idx+1), [(feature_names[i], topic[i].round(2)) for i in topic.argsort()[:-n - 1:-1]])  
6  
7 get_topics(lda_model.components_, terms)
```

```
Topic 1: [('government', 8658.95), ('queensland', 8134.58), ('perth', 6332.45), ('year', 5981.93), ('change', 5833.07)]  
Topic 2: [('world', 7026.33), ('house', 6217.97), ('donald', 5757.52), ('open', 5620.39), ('years', 5563.76)]  
Topic 3: [('police', 12140.34), ('kill', 6091.65), ('interview', 5921.12), ('live', 5657.67), ('rise', 4162.16)]  
Topic 4: [('court', 6173.46), ('crash', 5497.33), ('state', 4857.9), ('tasmania', 4443.89), ('accuse', 4300.92)]  
Topic 5: [('australia', 13994.07), ('south', 6253.18), ('woman', 5614.31), ('coast', 5465.23), ('warn', 5155.11)]  
Topic 6: [('charge', 8440.62), ('election', 7650.47), ('adelaide', 6839.75), ('murder', 6418.61), ('make', 6198.2)]  
Topic 7: [('help', 5372.6), ('miss', 4601.06), ('people', 4561.71), ('2016', 4212.58), ('family', 4149.3)]  
Topic 8: [('sydney', 8597.95), ('melbourne', 7603.52), ('canberra', 6285.91), ('plan', 5606.37), ('power', 4198.99)]  
Topic 9: [('attack', 6818.74), ('market', 5094.55), ('council', 3854.2), ('share', 3811.79), ('national', 3793.87)]  
Topic 10: [('trump', 13043.55), ('australian', 11389.92), ('north', 6241.08), ('school', 5726.25), ('report', 5560.48)]
```

---

# Word Cloud



# Word Cloud

- 단어를 출현 빈도에 비례하는 크기로 단어의 빈도수를 시각화하는 기법
- 빈도분석이 된 데이터를 시각화하기 위해 Word Cloud를 사용
- **pip install wordcloud**

# Sample을 이용한 Word Cloud 그리기

```
1 from konlpy.corpus import kolaw  
2 data = kolaw.open('constitution.txt').read()
```

```
1 from konlpy.tag import Komoran  
2 komoran = Komoran()
```

헌법 말뭉치를 불러와 형태  
소 분석 후 명사만 추출함

```
1 print(komoran.nouns("%r"%data[0:1000]))
```

```
['대한민국', '헌법', '유구', '한', '역사', '전통', '국민', '운동', '건  
립', '대한민국', '임시', '정부', '법통', '불의', '항거', '민주', '이념',  
'계승', '조국', '민주개혁', '평화', '통일', '사명', '입각', '정의', '인  
도', '동포애', '민족', '단결', '사회', '폐습', '불의', '타파', '자율',  
'조화', '바탕', '자유', '민주', '기본', '질서', '정치', '경제', '사회',  
'문화', '영역', '각인', '기회', '능력', '최고', '도로', '발휘', '자유',  
'권리', '책임', '의무', '완수', '안', '국민', '생활', '균등', '향상',  
'밖', '항구', '세계', '평화', '인류', '공영', '이바지', '우리들의', '자  
소', '아저', '자유', '행복', '화부', '거', '다진', '녀', '7월 12일', '제
```

# Sample을 이용한 Word Cloud 그리기 (Cont.)

```
1 word_list = komoran.nouns("%r \"%data[0:1000])
```

명사들을 공백으로 이어서  
하나의 문장으로 만듬

```
1 text = ' '.join(word_list)
```

```
1 text
```

'대한민국 헌법 유구 한 역사 전통 국민 운동 건립 대한민국 임시 정부 법통  
불의 항거 민주 이념 계승 조국 민주개혁 평화 통일 사명 입각 정의 인도 동  
포애 민족 단결 사회 폐습 불의 타파 자율 조화 바탕 자유 민주 기본 질서 정  
치 경제 사회 문화 영역 각인 기회 능력 최고 도로 발휘 자유 권리 책임 의무  
완수 안 국민 생활 균등 향상 밖 항구 세계 평화 인류 공영 이바지 우리들의  
자손 안전 자유 행복 확보 것 다짐 년 7월 12일 제정 차 개정 헌법 국회 의결  
국민 투표 개정 장 강 대한민국 민주공화국 대한민국 주권 국민 권력 국민 대  
한민국 국민 요건 법률 국가 법률 바 재외국민 보호 의무 대한민국 영토 한반  
도 부속 도서 대한민국 통일 지향 자유 민주 기본 질서 입각 평화 통일 정책  
수립 추진 대한민국 국제 평화 유지 노력 침략 전쟁 부인 국군 국가 안전 보  
장 국토방위 신성 의무 수행 사명 정치 중립 준수 헌법 체결 공포 조약 일반  
승인 국제 법규 국내법 효력 외국인 국제법 조약 바 지위 보장 공무원 국민  
전체 봉사자 국민 책임 공무원 신분 정치 중립 법률 바 보장 정당 설립 자유  
복수 정당'

# Sample을 이용한 Word Cloud 그리기 (Cont.)

```
1 import matplotlib.pyplot as plt  
2 %matplotlib inline  
3 from wordcloud import WordCloud
```

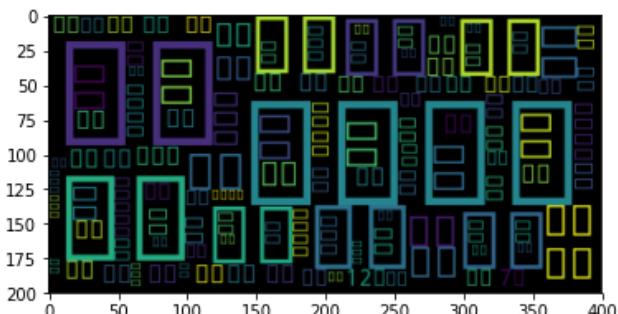
```
1 wordc = WordCloud()  
2 wordc.generate(text)
```

한글이 깨짐

```
<wordcloud.wordcloud.WordCloud at 0x1d532a20>
```

```
1 plt.figure()  
2 plt.imshow(wordc, interpolation="bilinear")
```

```
<matplotlib.image.AxesImage at 0x1f2da6d8>
```



# 한글 처리

```
1 wordc = WordCloud(background_color='white', max_words=20,  
2 font_path='c:/Windows/Fonts/malgun.ttf',  
3 relative_scaling=0.2)
```

```
1 wordc.generate(text)
```

```
<wordcloud.wordcloud.WordCloud at 0x1d536cc0>
```

```
1 plt.figure()  
2 plt.imshow(wordc, interpolation="bilinear")  
3 plt.axis('off')
```



# 전체 데이터를 이용한 Word Cloud 생성

```
1 word_list = komoran.nouns("%r %data")
2 text = ' '.join(word_list)
```

```
1 wordcloud = WordCloud(background_color='white', max_words=2000,  
2                         font_path='c:/Windows/Fonts/malgun.ttf',  
3                         relative_scaling=0.2)
```

```
1 wordcloud.generate(text)
```

```
<wordcloud.wordcloud.WordCloud at 0x203626a0>
```

```
1 plt.figure(figsize=(15, 10))  
2 plt.imshow(wordcloud, interpolation="bilinear")  
3 plt.axis('off')
```



# 불용어 사전 추가

```
1 from wordcloud import STOPWORDS  
2 from sklearn.feature_extraction.stop_words import ENGLISH_STOP_WORDS  
3  
4 불용어 = STOPWORDS | ENGLISH_STOP_WORDS | set(["대통령", "국가"])
```

```
1 wordcloud = WordCloud(background_color='white', max_words=2000,  
2                      stopwords=불용어,  
3                      font_path='c:/Windows/Fonts/malgun.ttf',  
4                      relative_scaling=0.2)  
5 wordcloud.generate(text)
```

```
<wordcloud.wordcloud.WordCloud at 0x204a2f28>
```

```
1 plt.figure(figsize=(15, 10))  
2 plt.imshow(wordcloud, interpolation="bilinear")  
3 plt.axis('off')
```



“대통령”, “국가” 단어를 불용어로 지정

# Masking

```
1 from PIL import Image          http://coderby.com/img/16
2 import numpy as np
3 img = Image.open("south_korea.png").convert("RGBA")
4 mask = Image.new("RGB", img.size, (255,255,255))
5 mask.paste(img, img)
6 mask = np.array(mask)
```

- ✓ Word Cloud를 지정된 Mask Image에 맞도록 표시
  - ✓ Mask Image는 <http://coderby.com/img/16>

```
1 wordcloud = WordCloud(background_color='white', max_words=2000,  
2                     font_path='C:/Windows/Fonts/malgun.ttf',  
3                     mask=mask, random_state=42)  
4 wordcloud.generate(text)  
5 wordcloud.to_file("result1.png")
```

# Pallet 변경

```
1 import random
2 def grey_color(word, font_size, position, orientation,
3                 random_state=None, **kwargs):
4     return 'hsl(0, 0%%, %d%%)' % random.randint(50, 100)
%표현
```

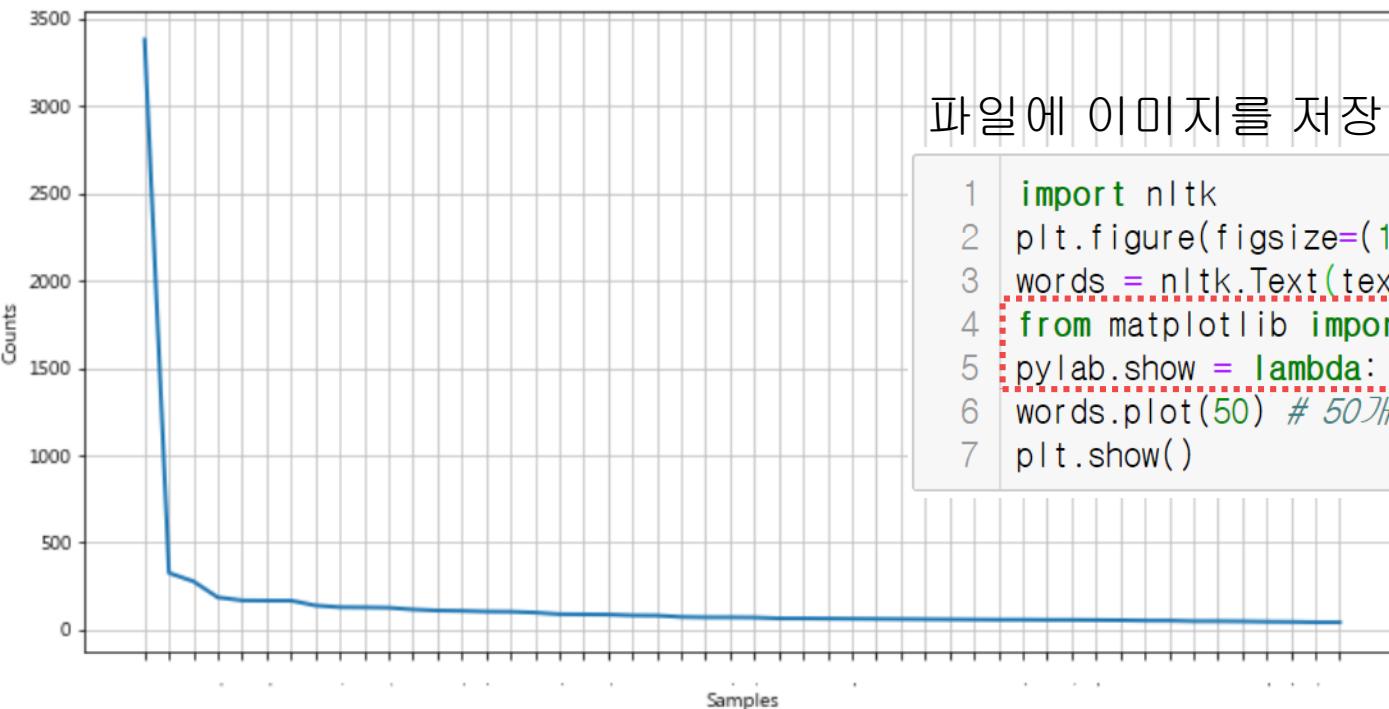
파일은 <http://coderby.com/img/15>에서



```
1 from PIL import Image
2 img = Image.open("south_korea_4x.png").convert("RGBA")
3 mask = Image.new("RGB", img.size, (255,255,255))
4 mask.paste(img, img)
5 mask = np.array(mask)
6 wordcloud = WordCloud(background_color='black', max_words=2000,
7                         font_path='C:/Windows/Fonts/malgun.ttf',
8                         mask=mask, random_state=42)
9 wordcloud.generate(text)
10 wordcloud.recolor(color_func=grey_color, random_state=3)
11 wordcloud.to_file("result2.png")
```

# 단어 빈도수 계산

```
1 import nltk  
2  
3 plt.figure(figsize=(12,6))  
4 words = nltk.Text(text)  
5 words.plot(50) # 50개만  
6 plt.show()
```



파일에 이미지를 저장하고 싶을 경우...

```
1 import nltk  
2 plt.figure(figsize=(12,6))  
3 words = nltk.Text(text)  
4 from matplotlib import pylab  
5 pylab.show = lambda: pylab.savefig('word_count.png')  
6 words.plot(50) # 50개만  
7 plt.show()
```



# Association Analysis

오렌지주스를 구매하는 사람은 와인을 구매할까?

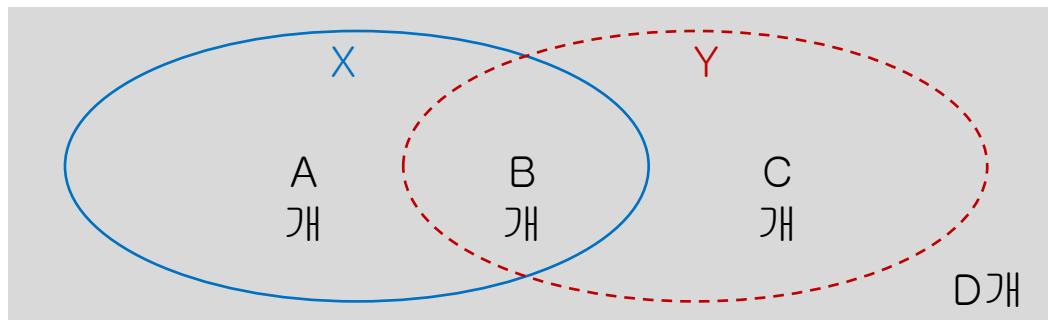
- 1 소주, 콜라, 와인
- 2 소주, 오렌지주스, 콜라
- 3 콜라, 맥주, 와인
- 4 소주, 콜라, 맥주
- 5 오렌지주스, 와인

# 연관 분석

- 연관 분석(Association Analysis)은 데이터들 사이에서 '자주 발생하는 속성을 찾는' 그리고 그 속성들 사이에 '연관성이 어느 정도 있는지'를 분석하는 방법
- 연관 규칙을 찾는 알고리즘으로는 apriori, FP-growth, DHP 알고리즘 등이 있음
- aprioir 알고리즘이 비교적 구현이 간단하고 성능 또한 높음  
`pip install apyori`

# 연관 분석 평가

- 지지도(Support)는 전체 Transaction에서 항목 집합(X, Y)을 포함하는 Transaction의 비율을 의미
- 신뢰도(Confidence)는 X를 포함하는 Transaction 중 Y도 포함하는 Transaction의 비율을 의미



범례 : ● X를 포함하는 트랜잭션

○ Y를 포함하는 트랜잭션

$$\text{지지도} = B\text{개} / (A\text{개} + B\text{개} + C\text{개} + D\text{개})$$

$$\text{신뢰도} = B\text{개} / (A\text{개} + B\text{개})$$

# 연관 분석 평가 측도

평가 측도	설명
지지도 (Support)	<ul style="list-style-type: none"><li>. 연관 규칙이 얼마나 중요한지에 대한 평가</li><li>. 낮은 지지도의 연관 규칙은 우연히 발생한 트랜잭션에서 생성된 규칙일 수 있음</li></ul>
신뢰도 (Confidence)	<ul style="list-style-type: none"><li>. 연관 규칙이 얼마나 믿을 수 있는지 평가</li></ul>
향상도 (Lift)	<ul style="list-style-type: none"><li>. <math>\text{Lift} = P(Y X) / P(Y) = P(X \cap Y) / P(X)P(Y) = \text{신뢰도} / P(Y)</math></li><li>. <math>X \rightarrow Y</math>의 신뢰도 / <math>Y</math>의 지지도</li><li>. 1 보다 작으면 음의 상관관계(설사약과 변비약), 1이면 상관관계 없음(과자와 후추), 1 보다 크면 양의 상관관계(빵과 버터)</li></ul>
파이 계수 (Phi coefficient)	<ul style="list-style-type: none"><li>. -1 : 완전 음의 상관관계, 0 : 상관관계 없음, 1 : 완전 양의 상관관계</li></ul>
IS 측도 (Interest-Support)	<ul style="list-style-type: none"><li>. 비대칭적 이항 변수에 적합한 측도로 값이 클수록 상관관계가 높음</li><li>. <math>P(X, Y) / P(X) * P(Y)</math></li></ul>

# CSV file로부터 Transaction Data 생성

- 트랜잭션(Transaction)은 서로 관련 있는 데이터의 모음
- 연관분석 시 Transaction으로부터 연관 규칙을 도출
- Python의 Transaction Data는 list형식

```
1 import csv
2 with open('basket.csv', 'r', encoding='UTF8') as cf:
3     transactions = []
4     r = csv.reader(cf)
5     for row in r:
6         transactions.append(row)
7 
```

```
1 transactions
```

```
[['소주', '콜라', '와인'],
 ['소주', '오렌지주스', '콜라'],
 ['콜라', '맥주', '와인'],
 ['소주', '콜라', '맥주'],
 ['오렌지주스', '와인']]
```

basket.csv

1	소주,콜라,와인
2	소주,오렌지주스,콜라
3	콜라,맥주,와인
4	소주,콜라,맥주
5	오렌지주스,와인

# 연관 규칙 생성

- apriori() 연역적 알고리즘(A Priori Algorithm)을 사용하여 연관 규칙을 알아내는 함수

```
1 from apyori import apriori  
2 rules = apriori(transactions, min_support=0.1, min_confidence=0.1)  
3 results = list(rules)  
4 type(results)
```

list

```
1 results[0]
```

```
RelationRecord(items=frozenset({'맥주'}), support=0.4, ordered_statistics=[OrderedStatistic(items_base=frozenset(), items_add=frozenset({'맥주'}), confidence=0.4, lift=1.0)])
```

```
1 results[10]
```

```
RelationRecord(items=frozenset({'콜라', '소주'}), support=0.6, ordered_statistics=[OrderedStatistic(items_base=frozenset({'소주'}), items_add=frozenset({'콜라'}), confidence=1.0, lift=1.25), OrderedStatistic(items_base=frozenset({'콜라'}), items_add=frozenset({'소주'}), confidence=0.7499999999999999, lift=1.2499999999999998)])
```

이렇게 생성한 규칙은  
namedtuple 형식으로  
저장되어 있음

# 연관 규칙 형식

- 연관 규칙은 *namedtuple* 형식으로 저장되어 있기 때문에 원하는 값을 찾거나 출력하기 위해서는 연관 규칙 결과가 어떤 형식으로 저장되어 있는지 이해해야 함
- apyori.py 파일에 정의되어 있는 **RelationRecord**와 **OrderStatistic**은 다음처럼 *namedtuple*로 정의되어 있음

```
SupportRecord = namedtuple( 'SupportRecord', ('items', 'support'))
```

```
RelationRecord = namedtuple( 'RelationRecord', SupportRecord._fields + ('ordered_statistics',))
```

```
OrderedStatistic = namedtuple( 'OrderedStatistic', ('items_base', 'items_add', 'confidence', 'lift',))
```

C:\Users\사용자\Anaconda3\Lib\site-packages\apyori.py

# 연관 규칙 조회

```
1 print("lhs rhs support confidence lift")
2 for row in results:
3     support = row[1]
4     ordered_stat = row[2]
5     for ordered_item in ordered_stat:
6         lhs = [x for x in ordered_item[0]]
7         rhs = [x for x in ordered_item[1]]
8         confidence = ordered_item[2]
9         lift = ordered_item[3]
10        print(lhs, " => ", rhs, "支持度{:>5.4f}置信度{:>5.4f}提升度{:>5.4f}".
11                  format(support, confidence, lift))
12
```

lhs	rhs	support	confidence	lift
[] =>	['맥주']	0.4000	0.4000	1.0000
[] =>	['소주']	0.6000	0.6000	1.0000
[] =>	['오렌지주스']	0.4000	0.4000	1.0000
[] =>	['와인']	0.6000	0.6000	1.0000
[] =>	['콜라']	0.8000	0.8000	1.0000
['맥주'] =>	['소주']	0.2000	0.5000	0.8333
['소주'] =>	['맥주']	0.2000	0.3333	0.8333
['맥주'] =>	['와인']	0.2000	0.5000	0.8333

# 연관 규칙 평가

- 오렌지주스를 구매한 사람은 와인을 구매할까?
- 앞의 결과에서 오렌지주스와 와인의 연관관계를 보면 다음과 같음
  - ['오렌지주스'] => ['와인'], 0.2000, 0.5000, 0.8
  - ['와인'] => ['오렌지주스'], 0.2000, 0.3333, 0.8
- 오렌지주스와 와인을 구매한 사람은 순서에 상관없이 전체의 20%
- 오렌지주스를 산 고객의 50%가 와인을 구매
- 향상도는 0.833이고 향상도가 1보다 작은 것은 음의 상관관계를 의미
- 그러므로 오렌지주스를 구매한 사람은 와인을 구매하지 않음

# 뉴스 RSS 서버에서 링크 주소 가져오기

- JTBC 경제분야 Crawling 후 연관 분석
- RSS 주소: <http://fs.jtbc.joins.com/RSS/economy.xml>

```
1 import requests  
2 rss_url = "http://fs.jtbc.joins.com/RSS/economy.xml"  
3 jtbc_economy = requests.get(rss_url)
```

```
1 from bs4 import BeautifulSoup  
2 economy_news_list = BeautifulSoup(jtbc_economy.content, "xml")  
3 link_list = economy_news_list.select("item > link")
```

```
1 len(link_list)
```

20

```
1 link_list[0].text
```

'[http://news.jtbc.joins.com/article/article.aspx?news\\_id=NB11822526](http://news.jtbc.joins.com/article/article.aspx?news_id=NB11822526)'

# 기사 수집 및 형태소 분석

- 기사 원 글을 수집해서 기사의 본문 내용을 형태소 분석 후 명사만 뽑음

```
1 #!pip install konfpy
```

```
1 from konlpy.tag import Kkma  
2 kkma = Kkma()
```

# 연관 분석

## ■ 연관 규칙을 생성하고, DataFrame으로 만듦

```
1 from apyori import apriori  
2 rules = apriori(news, min_support=0.3, min_confidence=0.2)  
3 results = list(rules)
```

```
1 import pandas as pd  
2 result_df = pd.DataFrame(None,  
3                           columns=["lhs", "rhs", "support",  
4                                     "confidence", "lift"])  
5 index = 0  
6 for row in results:  
7     support = row[1]  
8     ordered_stat = row[2]  
9     for ordered_item in ordered_stat:  
10         lhs = " ".join(x.strip() for x in ordered_item[0])  
11         rhs = " ".join(x.strip() for x in ordered_item[1])  
12         confidence = ordered_item[2]  
13         lift = ordered_item[3]  
14         result_df.loc[index] = [lhs, rhs, support, confidence, lift]  
15         index = index + 1
```

# 연관 분석 탐색

## ■ 뉴스 기사 연관 분석

```
1 result_df.loc[(result_df.lhs.str.contains("택시")) &  
2                 (result_df.rhs=="공유")].sort_values(by=["lift"],  
3                                               ascending=False)
```

		lhs	rhs	support	confidence	lift
4084	서비스 대표 택시 앵커 업계	공유	0.3	1.000000	3.333333	
217	경제 택시	공유	0.3	1.000000	3.333333	
4402	혁신 이재웅 이재 택시 업계	공유	0.3	1.000000	3.333333	
4632	서비스 대표 경제 택시 앵커 업계	공유	0.3	1.000000	3.333333	
4653	서비스 이재 경제 택시 앵커 대표	공유	0.3	1.000000	3.333333	
4667	이재웅 서비스 경제 택시 앵커 대표	공유	0.3	1.000000	3.333333	
4681	혁신 서비스 경제 택시 앵커 대표	공유	0.3	1.000000	3.333333	