

# **Text Mining II**

**Bok, Jong Soon**  
[javaexpert@nate.com](mailto:javaexpert@nate.com)  
<https://github.com/swacademy>

# Contents

- Word Embedding
- Text Classification
- Tagging Task

---

# I. Word Embedding



---

# 1. Word Embedding



# Word Embedding

- 자연어를 컴퓨터가 이해하고, 효율적으로 처리하게 하기 위해서는 컴퓨터가 이해할 수 있도록 자연어를 적절히 변환할 필요가 있다.
- 단어를 표현하는 방법에 따라서 자연어 처리의 성능이 크게 달라지기 때문에 이에 대한 많은 연구가 있었고, 여러 가지 방법들이 알려져 있다.
- 단어를 벡터로 표현하는 대표적인 방법으로 주로 희소 표현에서 밀집 표현으로 변환하는 것을 의미한다.

# Sparse Representation

- Vector 또는 Matrix의 값이 대부분이 0으로 표현되는 방법.
- One-hot Vector
- DTM
- 문제점
  - 단어의 개수가 늘어나면 Vector의 차원이 한없이 커진다는 점.
  - 단어 집합이 클수록 고차원의 Vector가 된다.
  - 공간적 낭비
  - 단어의 의미를 담지 못한다.
- 예) 단어가 10,000개 있고 강아지란 단어의 Index는 5이다.  
강아지 = [0 0 0 0 0 1 0 0 0 0 ... 0] #1뒤에 0의 수는 9995개

# Dense Representation

- Vector의 차원을 단어 집합의 크기로 설정하지 않는다.
- 사용자가 설정한 값으로 모든 단어의 Vector 표현의 차원을 맞춤.
- 0과 1만 가진 값이 아닌 실수 값도 표현 가능.

예) 강아지 = [ 0 0 0 0 1 0 0 0 0 0 0 ... 0 ]

# 1 뒤의 0의 수는 9995개. 차원은 10,000

- 만일 사용자가 밀집 표현의 차원을 128로 설정한다면, 모든 단어의 Vector 표현의 차원은 128로 바뀌면서 모든 값이 실수가 된다.

예) 강아지 = [0.2 1.8 1.1 -2.1 1.1 2.8 ...] # 이 Vector의 차원은 128

- Vector의 차원이 조밀해졌음.

# Word Embedding Again.

- 단어를 밀집 벡터(Dense Vector)의 형태로 표현하는 방법
- *Embedding Vector*
- Word Embedding 방법론
  - LSA
  - Word2Vec
  - FastText
  - GloVe

# Word Embedding Again.(Cont.)

	One-hot Vector	Embedding Vector
차원	고차원(단어 집합의 크기)	저차원
다른 표현	Sparse Vector의 일종	Dense Vector의 일종
표현 방법	수동	훈련 Data로부터 학습
값 Type	1과 0	실수

---

## 2. Word2Vec

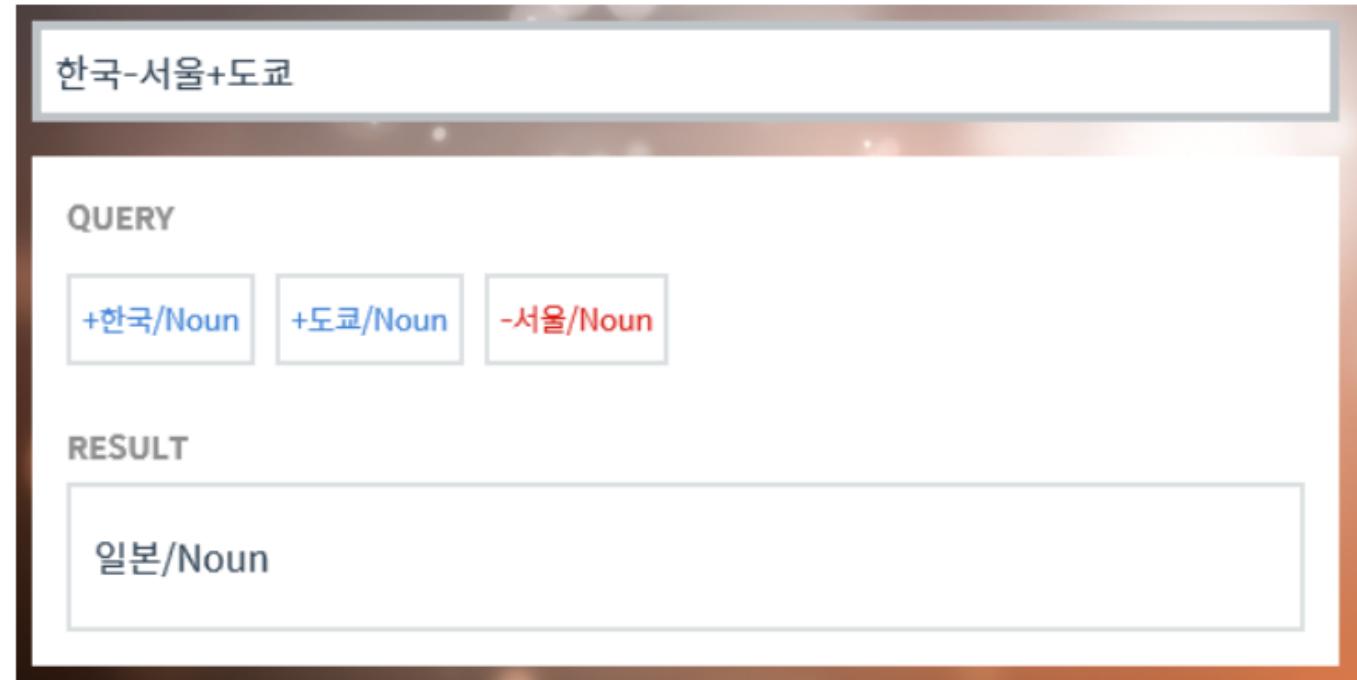


# Word2Vec

- One-hot Vector는 단어간 유사도를 계산할 수 없다는 단점이 있었음.

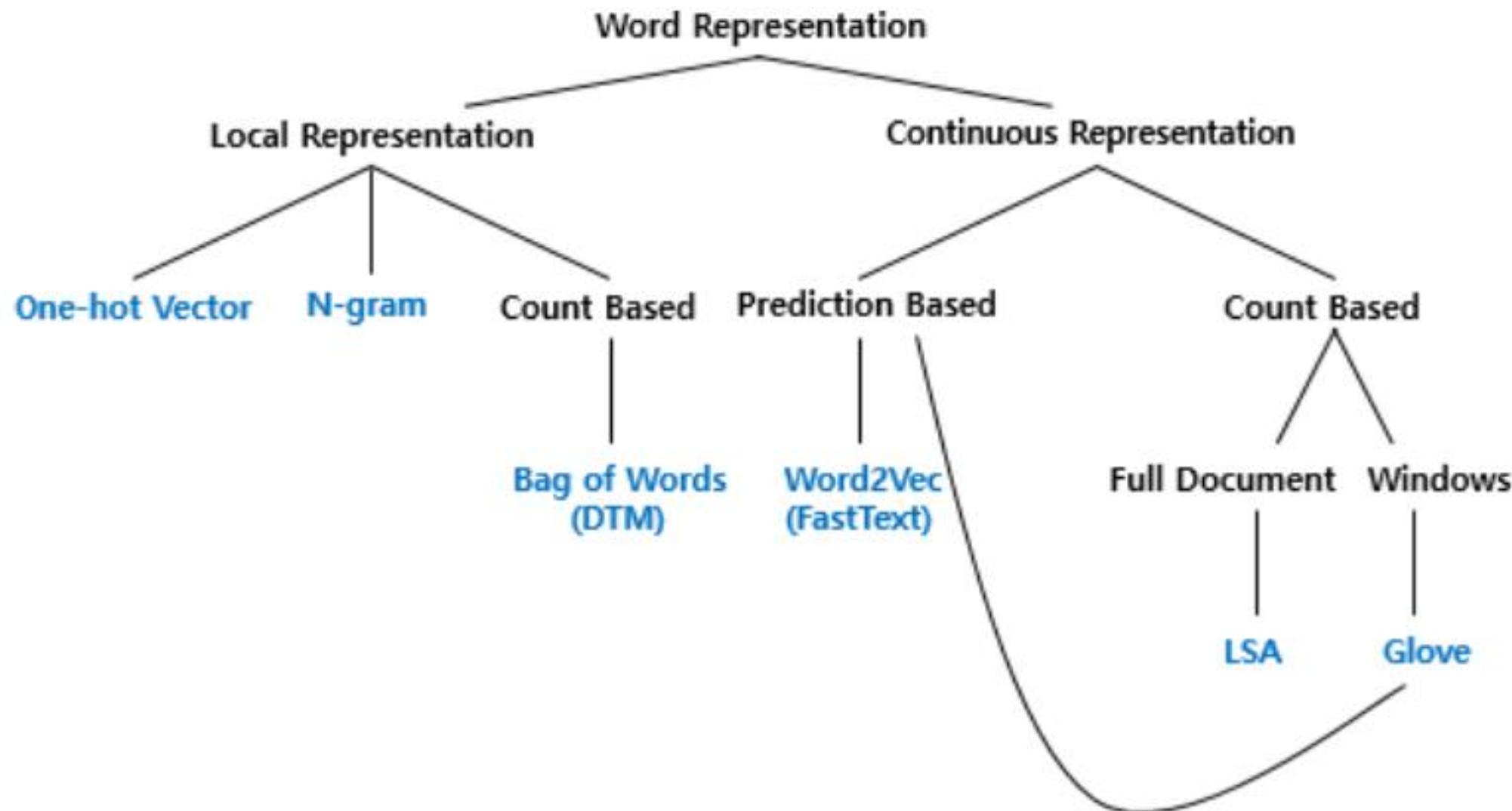
- <http://w.elnn.kr/search/>

- 고양이 + 애교 = 강아지
- 한국 - 서울 + 도쿄 = 일본
- 박찬호 - 야구 + 축구 = 호나우두



- 연산을 할 수 있는 이유는 각 단어에 있는 어떤 상징성, 의미가 Vector로 표현되었기 때문.
- Distributed Representation(Continuous Representation)
  - 단어의 **의미**를 다차원 공간에 Vector화 하는 방법

# 단어 표현의 Categories Again.



# Distributed Representation

- Distribution Hypothesis
  - 비슷한 위치에서 등장하는 단어들은 비슷한 의미를 가진다는 가정
  - 만일 강아지라는 단어가 귀엽다, 예쁘다, 애교 등의 단어가 주로 함께 등장한다면 이 Text를 Vector화할 때 의미적으로 가까운 단어가 된다.
- 분산 표현은 분포 가설을 이용하여 단어들의 Set을 학습하고, Vector에 단어의 의미를 여러 차원에 분산하여 표현.
- Vector의 차원이 곧 단어 집합(vocabulary)의 크기일 필요가 없음.
- Vector의 차원이 상대적으로 저 차원으로 줄어듦.
- 분산 표현은 저 차원에 단어의 의미를 여러 차원에다가 분산하여 표현.
- 단어 간 유사도 계산 가능.

# CBOW(Continuous Bag of Words)

- Word2Vec
  - CBOW(Continuous Bag of Words)
  - Skip-Gram
- CBOW
  - 주변에 있는 단어들을 가지고, 중간에 있는 단어들을 예측하는 방법
- Skip-Gram
  - 중간에 있는 단어를 가지고 주변 단어들을 예측하는 방법
- 두 Mechanism은 거의 같음.

## CBOW(Continuous Bag of Words) (Cont.)

- 예문 : **The fat cat sat on the mat.**
- {"The", "fat", "cat", "on", "the", "mat"}으로부터 sat을 예측하는 것
- 단어 **sat** → 중심단어(Center Word)
- 예측에 사용된 단어(**The, fat, cat, on, the, mat**) → 주변 단어(Context Word)
- Window : 중심 단어를 예측하기 위해 앞, 뒤로 몇 개의 단어를 볼 것인가에 대한 범위
- 예:window의 크기가 2이고, 예측하고자 하는 중심 단어가 sat 이면?
  - 앞의 두 단어인 fat, cat, 그리고 뒤에 있는 2단어 on, the를 참고한다.

# CBOW(Continuous Bag of Words) (Cont.)

- 즉, window의 크기가 n이면, 실제 중심 단어를 예측하기 위해 참조하려는 단어의 개수는  $2n$ 이다.

중심 단어  
↓  
**The fat cat** sat on the mat

The fat cat sat on the mat

The fat **cat** sat on the mat

The fat cat **sat** on the mat

The fat cat sat **on** the mat

The fat cat sat on **the** mat

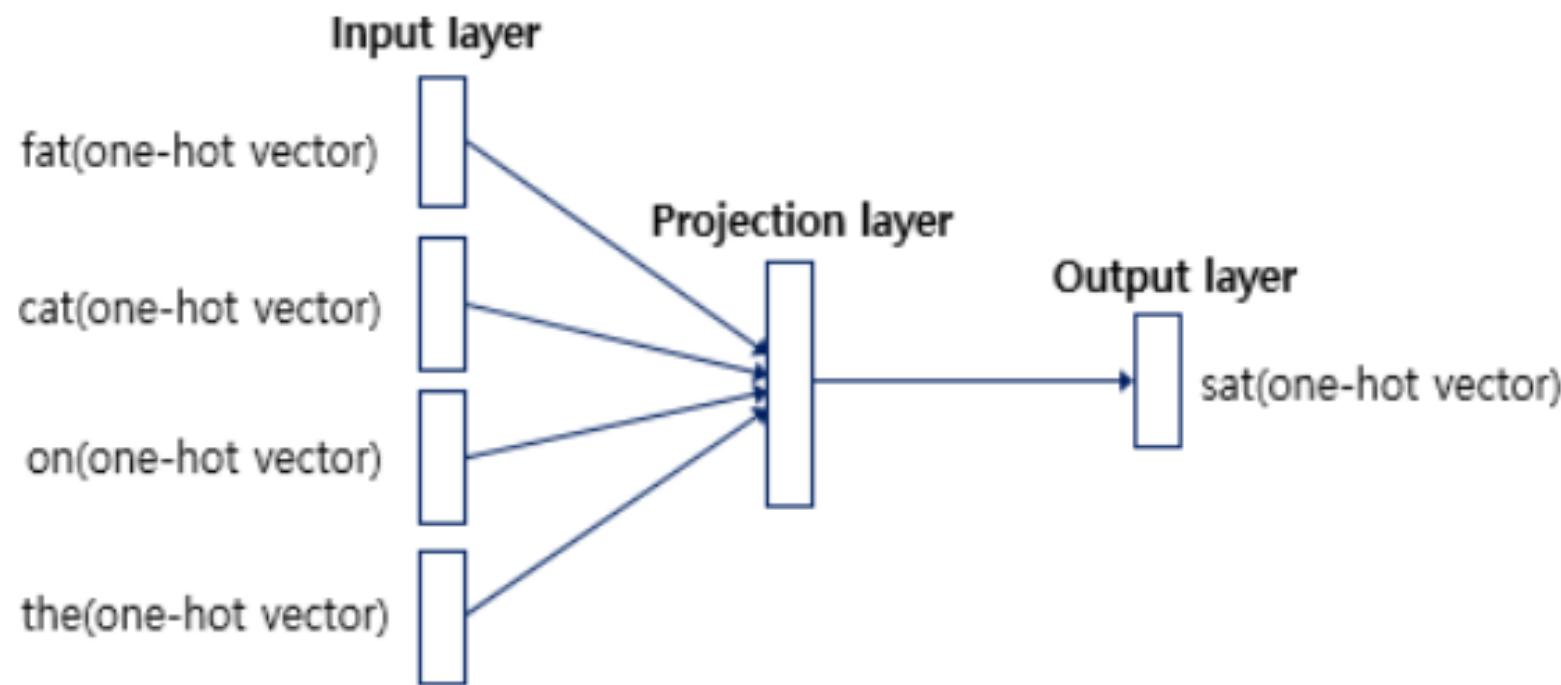
The fat cat sat on the **mat**

중심 단어	주변 단어
[1, 0, 0, 0, 0, 0, 0]	[0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0]	[0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0]	[0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0]	[0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 1]	[0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1]

# CBOW(Continuous Bag of Words) (Cont.)

## ■ Sliding Window

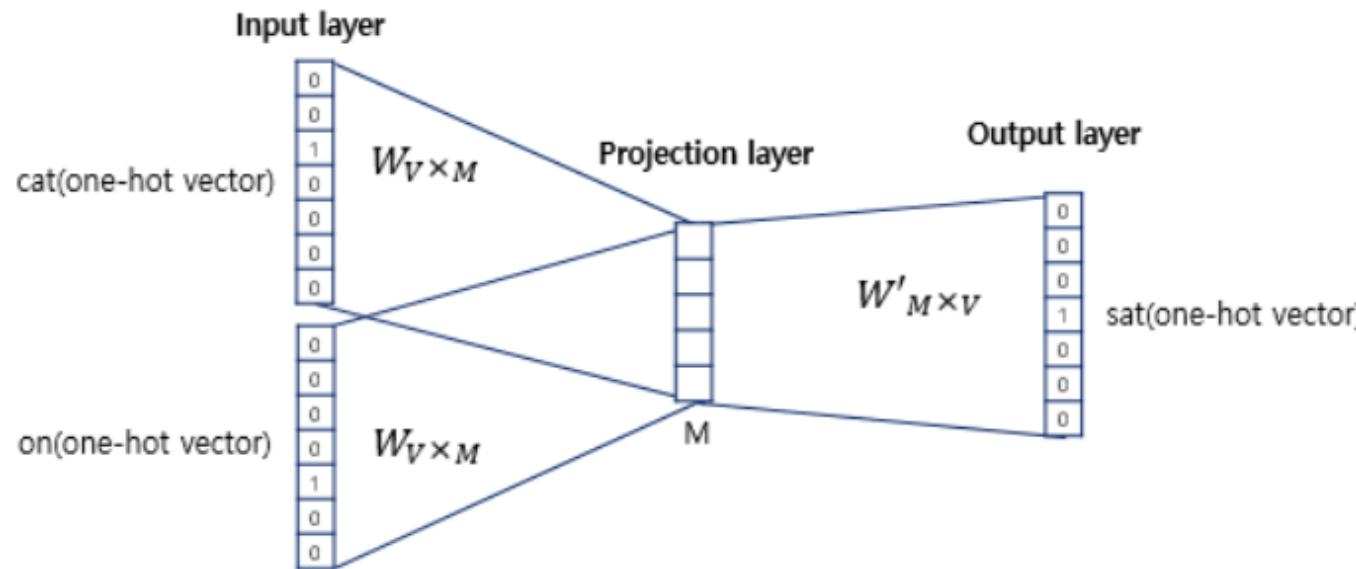
- window의 크기가 정해지면, window를 계속 움직이면서 주변 단어와 중심 단어 선택을 바꿔가며 학습을 위한 Dataset를 만들 수 있는 방법



# CBOW(Continuous Bag of Words) (Cont.)

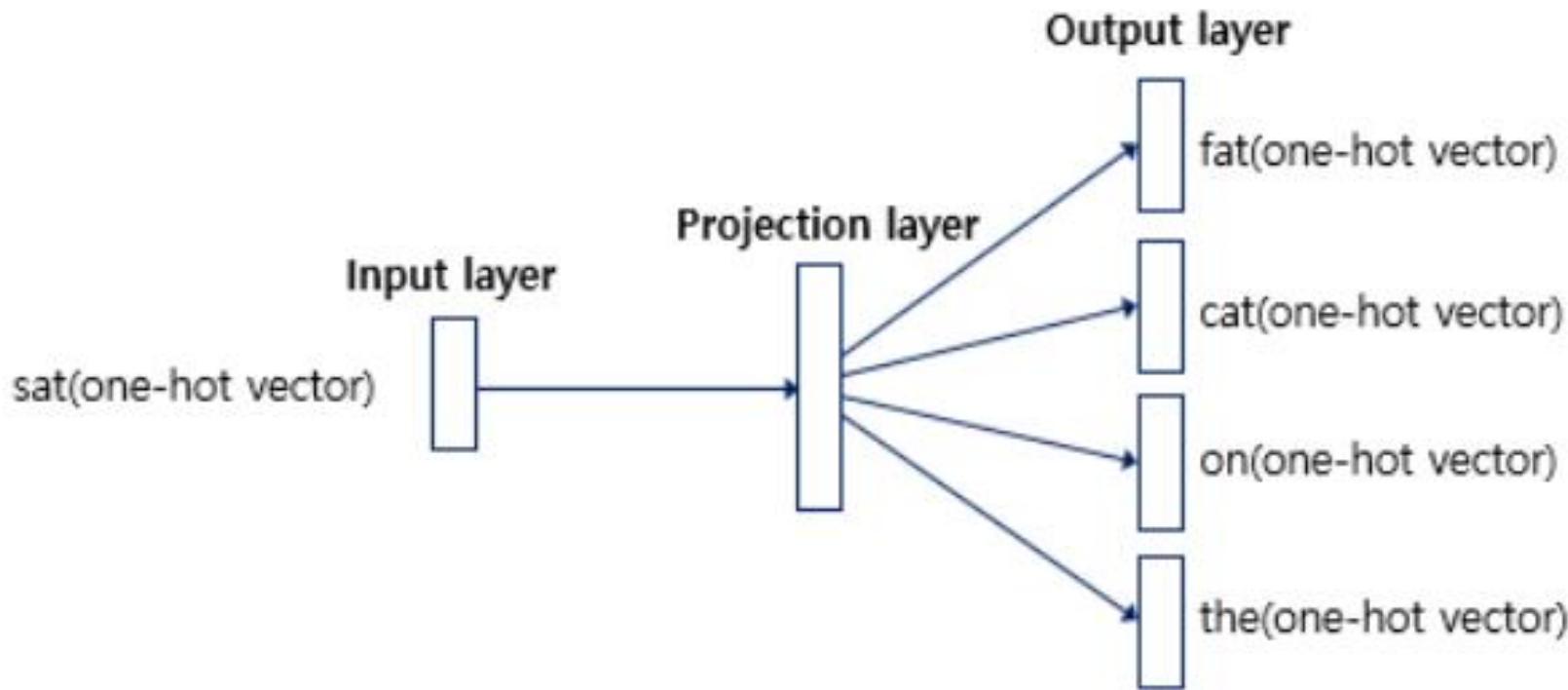
## ■ Word2Vec은 Deep Learning이 아니다.

- Deep Learning과 달리 입력층과 출력층 사이에 하나의 은닉층만 존재.
- 그래서 Deep Neural Network이 아니라 Shallow Neural Network이라 함.
- Word2Vec의 은닉층은 활성화함수가 존재하지 않음.
- Word2Vec의 은닉층 → 투사층(Projection Layer)
  - Lookup table이라는 연산을 담당하는 층



# Skip-gram

- 중심 단어에서 주변 단어 예측



- 전반적으로 Skip-gram이 CBOW보다 성능이 좋음.

# English Word2Vec 만들기

- \$ pip install genism

- Word2Vec를 위한 Corpus

- [https://wit3.fbk.eu/get.php?path=XML\\_releases/xml/ted\\_en-20160408.zip&filename=ted\\_en-20160408.zip](https://wit3.fbk.eu/get.php?path=XML_releases/xml/ted_en-20160408.zip&filename=ted_en-20160408.zip)

- ted\_en-20160408.xml

- <content>...</content>가 필요

- (Laughter)나 (Applause)와 같은 배경음을 나타내는 단어도 제거해야

```
<content>
Here are two reasons companies fail: they only do more of the same, or they only do what's new.
To me the real, real solution to quality growth is figuring out the balance between two activities: exploration and
exploitation. Both are necessary, but it can be too much of a good thing.
Consider Facit. I'm actually old enough to remember them. Facit was a fantastic company. They were born deep in the
Swedish forest, and they made the best mechanical calculators in the world. Everybody used them. And what did Facit
do when the electronic calculator came along? They continued doing exactly the same. In six months, they went from
maximum revenue ... and they were gone. Gone.
To me, the irony about the Facit story is hearing about the Facit engineers, who had bought cheap, small electronic
calculators in Japan that they used to double-check their calculators.
```

# English Word2Vec 만들기 (Cont.)

```
1 import re
2 from lxml import etree
3 import nltk
4 from nltk.tokenize import word_tokenize, sent_tokenize
5
6 targetXML=open('ted_en-20160408.xml', 'r', encoding='utf-8')
7
8 target_text = etree.parse(targetXML)
9 parse_text = '\n'.join(target_text.xpath('//content/text()'))
10 # xml 파일로부터 <content>와 </content> 사이의 내용만 가져온다.
11
12 content_text = re.sub(r'#[^"]*"', '', parse_text)
13 # 정규 표현식의 sub 모듈을 통해 content 중간에 등장하는 (Audio), (Laughter) 등의 배경을 부분을 제거.
14 # 해당 코드는 괄호로 구성된 내용을 제거.
15
16 sent_text=sent_tokenize(content_text)
17 # 입력 코퍼스에 대해서 NLTK를 이용하여 문장 토큰화를 수행.
18
19 normalized_text = []
20 for string in sent_text:
21     tokens = re.sub(r"[^a-zA-Z]+", " ", string.lower())
22     normalized_text.append(tokens)
23 # 각 문장에 대해서 구두점을 제거하고, 대문자를 소문자로 변환.
```

# English Word2Vec 만들기 (Cont.)

```
25 result=[]
26 result=[word_tokenize(sentence) for sentence in normalized_text]
27 # 각 문장에 대해서 NLTK를 이용하여 단어 토큰화를 수행.
28
29 print(result[:10])
30 # 문장 10개만 출력
```

```
[['here', 'are', 'two', 'reasons', 'companies', 'fail', 'they', 'only', 'do', 'more', 'of', 'the', 'same',
'or', 'they', 'only', 'do', 'what', 's', 'new'], ['to', 'me', 'the', 'real', 'real', 'solution',
'to', 'quality', 'growth', 'is', 'figuring', 'out', 'the', 'balance', 'between', 'two', 'activities',
'exploration', 'and', 'exploitation'], ['both', 'are', 'necessary', 'but', 'it', 'can', 'be', 'too',
'much', 'of', 'a', 'good', 'thing'], ['consider', 'facit'], ['i', 'm', 'actually', 'old', 'enough',
'to', 'remember', 'them'], ['facit', 'was', 'a', 'fantastic', 'company'], ['they', 'were', 'born',
'deep', 'in', 'the', 'swedish', 'forest', 'and', 'they', 'made', 'the', 'best', 'mechanical',
'calculators', 'in', 'the', 'world'], ['everybody', 'used', 'them'], ['and', 'what', 'did', 'facit', 'do',
'when', 'the', 'electronic', 'calculator', 'came', 'along'], ['they', 'continued', 'doing', 'exactly',
'the', 'same']]
```

# English Word2Vec 만들기 (Cont.)

```
1 from gensim.models import Word2Vec  
2 model = Word2Vec(sentences=result, size=100, window=5, min_count=5, workers=4, sg=0)
```

## ■ Word2Vec() parameters

- **size**

- Word Vector의 특징 값
- Embedding 된 Vector의 차원

- **window**

- Context window 크기

- **min\_count**

- 단어 최소 빈도 수 제한(빈도가 적은 단어들은 학습하지 않음.)

- **workers**

- 학습을 위한 Process 수

- **sg**

- 0(CBOW), 1(Skip-gram)

# English Word2Vec 만들기 (Cont.)

```
1 a=model.wv.most_similar("man")  
2 print(a)
```

```
[('woman', 0.852196455001831), ('guy', 0.8054466247558594), ('lady', 0.7874810695648193), ('boy', 0.762076  
4970779419), ('girl', 0.7238889932632446), ('soldier', 0.7014749050140381), ('gentleman', 0.69653773307800  
29), ('kid', 0.6960955858230591), ('poet', 0.6689803600311279), ('writer', 0.6524746417999268)]
```

- 유사 단어 찾기
  - man과 유사한 단어로 woman, guy, boy, lady, girl, gentlemen, soldier, kid, friend를 찾았음.
- 따라서 Word2Vec을 통해 단어의 유사도를 계산할 수 있게 되었음.

# Korean Word2Vec 만들기

## ■ Source

- Wikipedia 한국어 Dump file

## ■ 주의할 점

- 한국어는 前처리가 중요하여 Token化 할 때 띄어쓰기 단위가 아니라 형태소 단위로 token화 해야 embedding 성능이 좋아짐

## 1. Wikipedia 한국어 Dump File Downloads

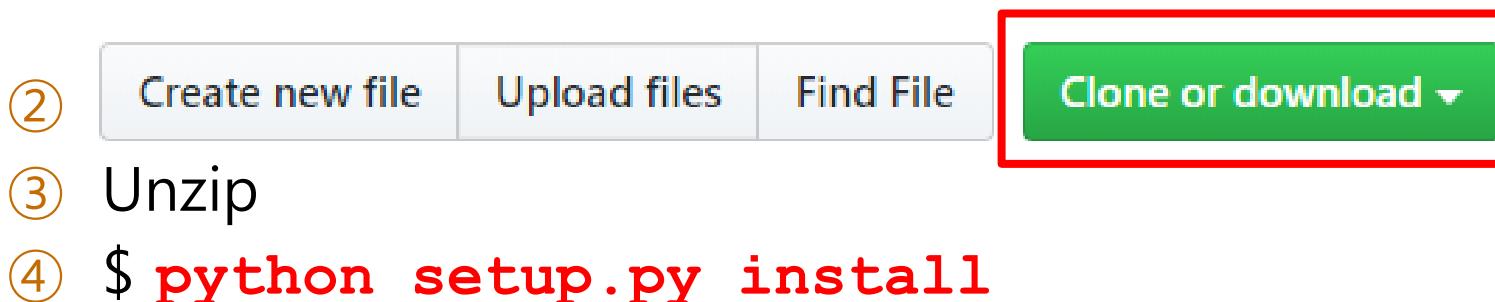
- <https://dumps.wikimedia.org/kowiki/latest/>

<a href="#">kowiki-latest-pages-articles-multistream.xml.bz2</a>	03-Sep-2019 01:24	685984729
<a href="#">kowiki-latest-pages-articles-multistream.xml.bz2-rss.xml</a>	12-Sep-2019 22:40	817
<a href="#">kowiki-latest-pages-articles.xml.bz2</a>	02-Sep-2019 14:18	632345292
<a href="#">kowiki-latest-pages-articles.xml.bz2-rss.xml</a>	12-Sep-2019 01:14	781
<a href="#">kowiki-latest-pages-logging.xml.gz</a>	12-Sep-2019 01:17	74903840
<a href="#">kowiki-latest-pages-logging.xml.gz-rss.xml</a>	12-Sep-2019 01:17	775

# Korean Word2Vec 만들기 (Cont.)

## 2. Wikipedia Extractor Downloads

- Downloads 받은 file은 xml file이기 때문에 Word2Vec을 원활하게 진행하려면 Text 형식의 file로 변환해야 함.
  - Wikipedia Dump file을 Text 형식으로 변환하는 Program
  - **git clone "https://github.com/attardi/wikiextractor.git"**
  - or
- ① <https://github.com/attardi/wikiextractor>



# Korean Word2Vec 만들기 (Cont.)

## 3. Wikipedia Dump file 변환

- Wikipedia Extractor와 Wikipedia 한국어 Dump file을 동일한 directory 경로에 두고, 아래 명령어를 실행하여 Text file로 변환(approximately 10 min.)
- \$ **python WikiExtractor.py kowiki-latest-pages-articles.xml.bz2**

```
INFO: 2563238 박민석 (2000년)
INFO: 2563241 이선우
INFO: 2563249 만수르 이사예프
INFO: 2563251 동키 킹
INFO: 2563253 명차현
INFO: 2563259 오혁철
INFO: 2563261 오윤경
INFO: 2563263 다곰바족
INFO: 2563264 엄철성
INFO: 2563265 오늘의 강론
INFO: 2563267 페르닐레
INFO: 2563268 안드리치
INFO: 2563269 박현일
INFO: 2563277 현승아
INFO: 2563281 온라인 백과사전 목록
INFO: 2563282 기치료
INFO: 2563283 기관 전속 위키백과 사용자
INFO: 2563285 한방언
INFO: 2563286 조성윤 (기업인)
INFO: 2563288 위키백과의 문서 훼손
INFO: Finished 3-process extraction of 467716 articles in 2085.7s (224.2 art/s)
INFO: total of page: 802888, total of articl page: 467716; total of used articl
page: 467716
```

# Korean Word2Vec 만들기 (Cont.)

## ■ 변환 후

- 변환된 Wikipedia 한국어 Dump는 총 7개의 directory
- AA ~ AG의 directory

```
instructor@MyDesktop:~/PythonHome/wikiextractor$ cd text
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ ls
AA  AB  AC  AD  AE  AF  AG
```

- 각 directory내에는 wiki\_00 ~ wiki\_90이라는 파일

```
<doc id="문서 번호" url="실제 위키피디아 문서 주소" title="문서 제목">
내용
</doc>
```

# Korean Word2Vec 만들기 (Cont.)

## ■ AA directory wiki\_00 file

```
instructor@MyDesktop: ~/PythonHome/wikiextractor/text/AA
File Edit View Search Terminal Help
-
```

</doc>  
<doc id="343" url="https://ko.wikipedia.org/wiki?curid=343" title="유리수">  
유리수  
  
수학에서, 유리수(有理數, )는 두 정수의 비율로 나타낼 수 있는 수이다. 단, 분모가 0이 아니어야 한다. 특히, 분모가 1일 수 있으므로 모든 정수는 유리수이다. 유리수체의 기호는 볼드체 formula\_1나 칠판 볼드체 formula\_2이며, '몫'을 뜻하는 영어()에서 따왔다.  
  
유리수체 formula\_2는 정수환 formula\_4의 분수체이다. 이는 다음과 같은 집합으로 생각할 수 있다.  
  
엄밀히 말해, 유리수체 formula\_2는 다음과 같은 공리를 만족시키는 (동형 아래 유일한) 체이다.  
  
유리수체 formula\_2는 구체적으로 다음과 같이 구성할 수 있다. 집합 formula\_11 위에 다음과 같은 동치 관계 formula\_12를 줄 수 있다.  
유리수체 formula\_2는 집합으로서 몫집합 formula\_15이며, 그 위의 덧셈과 곱셈은 다음과 같다.  
체가 만족시켜야 하는 조건인 각종 연산 법칙과 덧셈 항등원 formula\_18 및 각 유리수 formula\_19의 덧셈 역원 formula\_20 및 곱셈 항등원 formula\_21 및 0이 아닌 각 유리수 formula\_22의 곱셈 역원 formula\_23의 존재가 성립하므로, 이는 체를 이룬다.

# Korean Word2Vec 만들기 (Cont.)

## 4. 훈련 데이터 만들기

- AA directory 안의 모든 file wiki00 ~ wiki90 → wikiAA.txt

```
$ cat ./AA/wiki** >> wikiAA.txt
```

- 각 directory 동일하게 진행

```
$ cat ./AB/wiki** >> wikiAB.txt
```

```
$ cat ./AC/wiki** >> wikiAC.txt
```

```
$ cat ./AD/wiki** >> wikiAD.txt
```

```
$ cat ./AE/wiki** >> wikiAE.txt
```

```
$ cat ./AF/wiki** >> wikiAF.txt
```

```
$ cat ./AG/wiki** >> wikiAG.txt
```

- wikiAA.txt ~ wikiAG.txt

# Korean Word2Vec 만들기 (Cont.)

```
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat ./AB/wiki* >> wikiAB.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat ./AC/wiki* >> wikiAC.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat ./AD/wiki* >> wikiAD.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat ./AE/wiki* >> wikiAE.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat ./AF/wiki* >> wikiAF.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat ./AG/wiki* >> wikiAG.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ ls
AA AB AC AD AE AF AG wikiAA.txt wikiAB.txt wikiAC.txt wikiAD.txt wikiAE.txt wikiAF.txt wikiAG.txt
```

- 7개의 file을 하나의 file로 합치기

```
$ cat wiki*.txt >> wiki_data.txt
```

```
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ 
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ ls
AA AB AC AD AE AF AG wikiAA.txt wikiAB.txt wikiAC.txt wikiAD.txt wikiAE.txt wikiAF.txt wikiAG.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ 
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ cat wiki*.txt >> wiki_data.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ ls -l wiki_data.txt
-rw-r--r-- 1 instructor instructor 667273537 Sep 21 21:42 wiki_data.txt
instructor@MyDesktop:~/PythonHome/wikiextractor/text$ █
```

# Korean Word2Vec 만들기 (Cont.)

## 5. Word2Vec 작업

```
1 f = open(r'./wikiextractor/text/wiki_data.txt', encoding="utf8")
```

```
1 i=0
2 while True:
3     line = f.readline()
4     if line != '\n':
5         i=i+1
6         print("%d번째 줄 :" %i + line)
7     if i==5:
8         break
9 f.close()
```

1번째 줄 :<doc id="5" url="https://ko.wikipedia.org/wiki?curid=5" title="지미 카터">

2번째 줄 :지미 카터

3번째 줄 :제임스 얼 "지미" 카터 주니어(, 1924년 10월 1일 ~ )는 민주당 출신 미국 39번째 대통령 (1977년 ~ 1981년)이다.

4번째 줄 :지미 카터는 조지아주 셀터 카운티 플레인스 마을에서 태어났다. 조지아 공과대학교를 졸업하였다. 그 후 해군에 들어가 전함·원자력·잠수함의 승무원으로 일하였다. 1953년 미국 해군 대위로 예편하였고 이후 땅콩·면화 등을 가꿔 많은 돈을 벌었다. 그의 별명이 "땅콩 농부" (Peanut Farmer)로 알려졌다.

5번째 줄 :1962년 조지아 주 상원 의원 선거에서 낙선하나 그 선거가 부정선거였음을 입증하게 되어 당선되고, 1966년 조지아 주 지사 선거에 낙선하지만 1970년 조지아 주 지사를 역임했다. 대통령이 되기 전 조지아 주 상원 의원으로서 활동한 바 있다. 1971년부터 1975년까지 조지아 주 지사를 역임했다. 그 후에는 조지아 주 상원 의원으로서 활동한 바 있다.

# Korean Word2Vec 만들기 (Cont.)

## 5. Word2Vec 작업

```
1 from konlpy.tag import Okt
2 okt = Okt()
3 fread = open(r'./wikiextractor/text/wiki_data.txt', encoding="utf8")
4 # 파일을 다시 처음부터 읽음.
5 n=0
6 result = []
7
8 while True:
9     line = fread.readline() #한 줄씩 읽음.
10    if not line: break # 모두 읽으면 while문 종료.
11    n=n+1
12    if n%5000==0: # 5,000의 배수로 while문이 실행될 때마다 몇 번째 while문 실행인지 출력.
13        print("%d번째 While문."%n)
14    tokenlist = okt.pos(line, stem=True, norm=True) # 단어 토큰화
15    temp=[]
16    for word in tokenlist:
17        if word[1] in ["Noun"]:# 명사일 때만
18            temp.append((word[0])) # 해당 단어를 저장함
19
20    if temp: # 만약 이번에 읽은 데이터에 명사가 존재할 경우에만
21        result.append(temp) # 결과에 저장
22 fread.close()
```

# Korean Word2Vec 만들기 (Cont.)

## 5. Word2Vec 작업

- 여기서는 형태소 분석기로 KoNLPy의 *Okt*를 사용하였다.
- 한국어 Word2Vec을 할 때 가장 간단한 전처리 방법은 noise가 섞인 데이터로부터 한국어 명사만을 최대한 정확도가 높게 뽑아내는 형태소 분석기를 사용하는 것이다.
- *Okt*는 괜찮다(Not Mecab).
- 해당 코드를 통해 학습을 위한 데이터를 만드는 시간은 기본적으로 몇 시간 (approximately 5 hours +)이 걸린다.

# Korean Word2Vec 만들기 (Cont.)

## 5. Word2Vec 작업

- 학습 데이터의 길이

```
1 len(result)
```

- 약 260만여개의 line이 Nous Tokenization이 되어 저장되어 있는 상태이다.
- 이를 Word2Vec으로 학습시킨다.

```
1 from gensim.models import Word2Vec  
2 model = Word2Vec(result, size=100, window=5, min_count=5, workers=4, sg=0)
```

# Korean Word2Vec 만들기 (Cont.)

## 6. 입력단어로 유사한 단어들을 구해본다.

```
1 a=model.wv.most_similar("대한민국")
2 print(a)
```

```
[('한국', 0.6331368088722229), ('우리나라', 0.5405941009521484), ('조선민주주의인민공화국', 0.5400398969650269), ('정보통신부', 0.49965575337409973), ('고용노동부', 0.49638330936431885), ('경남', 0.47878748178482056), ('국내', 0.4761977791786194), ('국무총리실', 0.46891751885414124), ('공공기관', 0.46730121970176697), ('관세청', 0.46708711981773376)]
```

```
1 b=model.wv.most_similar("어벤져스")
2 print(b)
```

```
[('스파이더맨', 0.8560965657234192), ('아이언맨', 0.8376990556716919), ('데어데블', 0.7797115445137024), ('인크레더블', 0.7791407108306885), ('스타트렉', 0.7752881050109863), ('엑스맨', 0.7738450765609741), ('슈퍼맨', 0.7715340852737427), ('어벤져스', 0.7453964948654175), ('슈퍼히어로', 0.7452991008758545), ('다크나이트', 0.7413955926895142)]
```

# Pre-trained Word2vec embedding 사용하기

## ■ NLP 작업시 Embedding Vector를 얻고 싶을 때

- Keras의 Embedding()를 사용하여 갖고 있는 훈련 Data를 가지고 단어 Vector를 학습하는 방식으로 만드는 방법
- 사전에 훈련된 워드 임베딩(Pre-trained Word Embedding)를 가지고 원하는 작업에 사용하는 방법
- 예) Wikipedia 등과 같은 데이터를 Word2Vec이나 Glove 등으로 사전에 학습시켜놓은 Embedding Vector들을 이용.

# Pre-trained Word2vec embedding 사용하기 (Cont.)

## -English

### ■ Google's Pre-trained Word2vec 사용하기

- 약 300백만 개의 단어 Vector
- 각 Vector의 크기는 300
- Model Downloads :  
<https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit>
- File size : zipped(1.5GB), unzip(3.3GB)

# Pre-trained Word2vec embedding 사용하기 (Cont.)

## -Korean

### ■ 박규병님의 Word2vec

- Github : <https://github.com/Kyubyong/wordvectors>
- Downloads :  
<https://drive.google.com/file/d/0B0ZXk88koS2KbDhXdWg1Q2RydlU/view>
- File size : zipped(ko.zip, 80.6MB), unzip(ko.bin, 50.7MB)

```
1 import gensim  
2 model = gensim.models.Word2Vec.load('./ko.bin')
```

```
1 a=model.wv.most_similar("강아지")  
2 print(a)
```

```
[('고양이', 0.7290452718734741), ('거위', 0.7185635566711426), ('토끼', 0.7056223750114441), ('멧돼지', 0.6950401067733765), ('엄마', 0.693433403968811), ('난쟁이', 0.6806551218032837), ('한마리', 0.677029550075531), ('아가씨', 0.675035297870636), ('아빠', 0.6729635000228882), ('목걸이', 0.6512460708618164)]
```

# Word2Vec 다양한 사용 사례

- [https://brunch.co.kr/@goodvc78/16?fbclid=IwAR1QZZAeZe\\_tNWxnxVCRwl8PlouBPAaqSIJ1IBxJ-EKtfDfmLehi1MUV\\_Lk](https://brunch.co.kr/@goodvc78/16?fbclid=IwAR1QZZAeZe_tNWxnxVCRwl8PlouBPAaqSIJ1IBxJ-EKtfDfmLehi1MUV_Lk)

# Word2Vec의 장단점

## ■ 장점

- 단어간의 유사도 측정에 용이
- 단어간의 관계 파악에 용이
- Vector 연산을 통한 추론이 가능(예:한국 - 서울 + 도쿄 = ?)

## ■ 단점

- 단어의 subword information 무시(예:**서울** vs **서울시** vs **고양시**)
- Out Of Vocabulary(OOV)에서 적용 불가능

---

## 3. fastText



# fastText

- Facebook research에서 공개한 Open Source Library
- <https://research.fb.com/fasttext/>
- <https://fasttext.cc/>
- Training
  - 기존의 Word2Vec과 유사하나, 단어를 **n-gram**으로 나누어 학습을 수행
  - **n-gram**의 범위가 2 ~ 5일 때, 단어를 다음과 같이 분리하여 학습함.  
*"assumption"* = {as, ss, su, ...., ass, ssu, sum, ump, mpt, ...., ption, *assumption*}
  - 이 때, **n-gram**으로 나눠진 단어는 사전에 들어가지 않으며, 별도의 n-gram vector를 형성

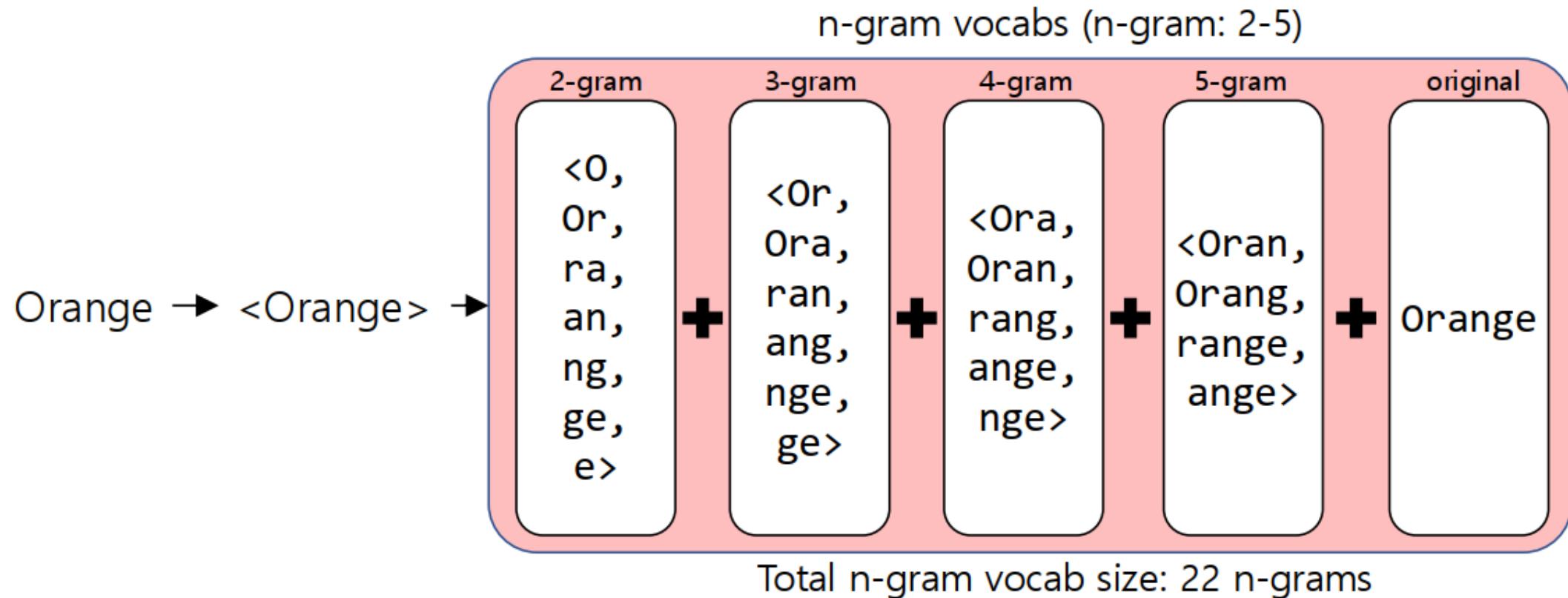
# fastText (Cont.)

## ■ Testing

- 입력 단어가 vocabulary에 있을 경우, Word2Vec 같은 방식으로 해당 단어의 Word Vector를 Return함.
- 만약 OOV일 경우, 입력 단어의 n-gram vector들의 합산을 Return함.

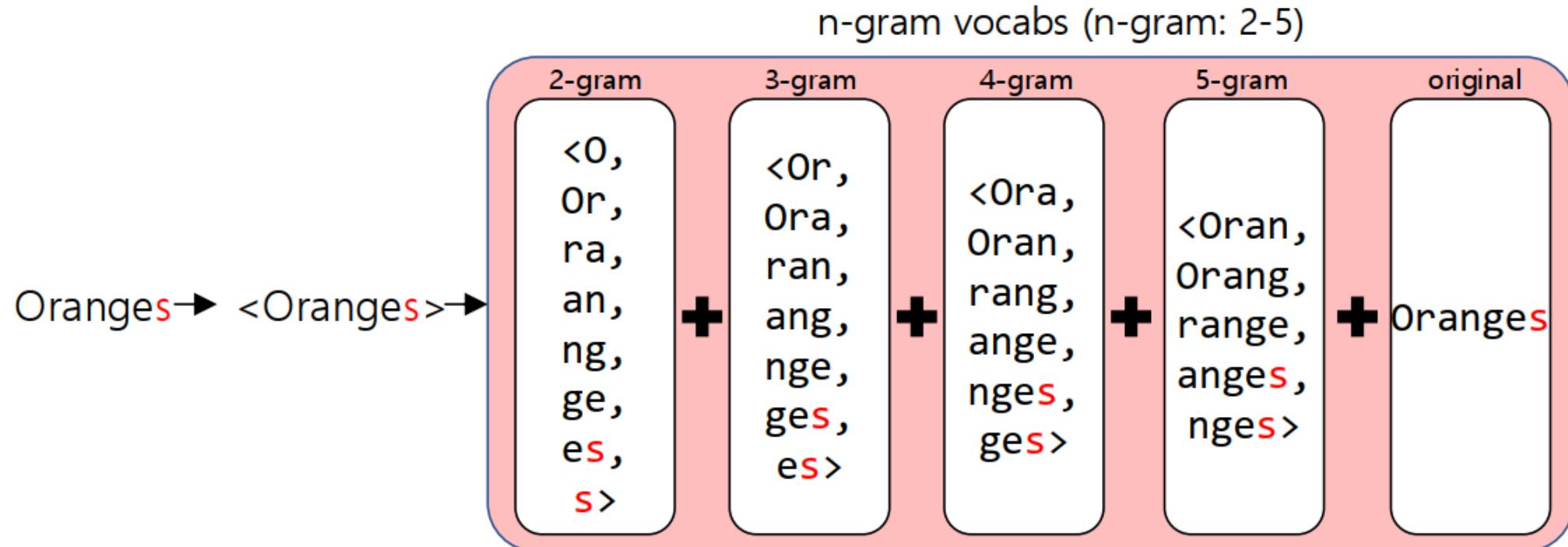
## fastText (Cont.)

- 단어를 n-gram으로 분리한 후, 모든 n-gram vector를 합산한 후 평균을 통해 단어 Vector를 획득



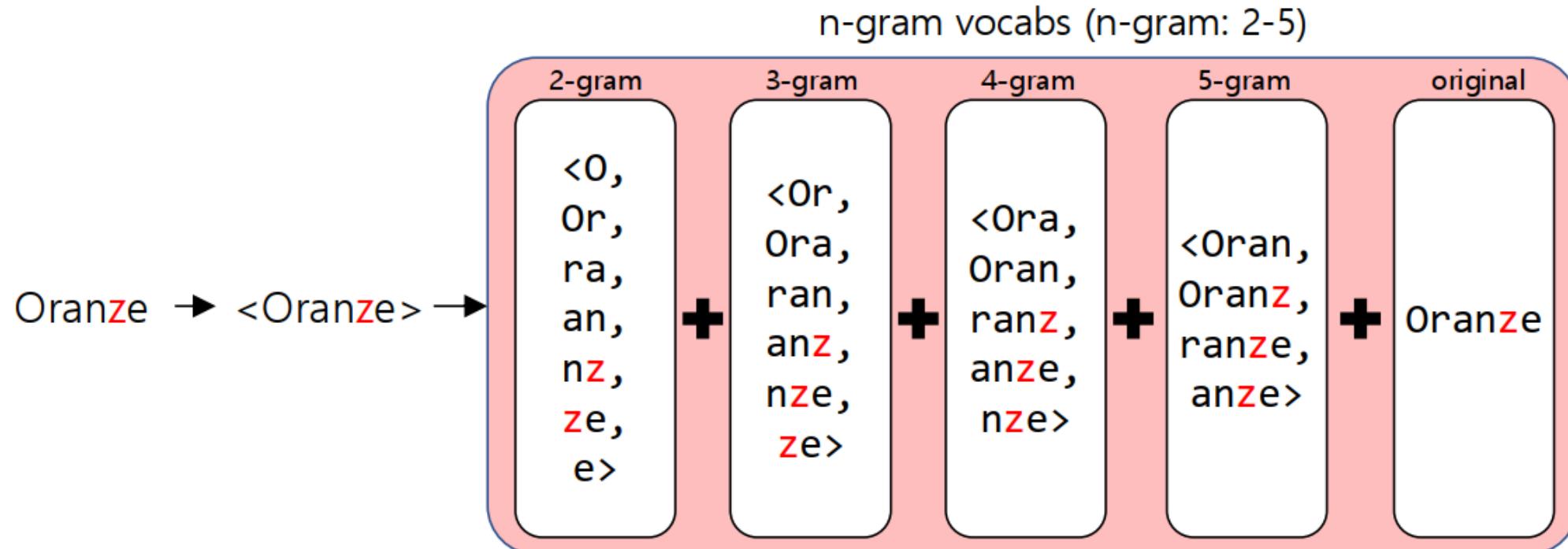
## fastText (Cont.)

- 단어를 n-gram으로 분리한 후, 모든 n-gram vector를 합산한 후 평균을 통해 단어 Vector를 획득



## fastText (Cont.)

- 심지어 오탈자 입력에 대해서도 본래 단어와 유사한 n-gram이 많아, 유사한 단어 Vector 획득 가능.



# fastText (Cont.)

## ■ 오탈자, OOV, 등장 횟수가 적은 학습 단어에 대해 강세

Input word	DF	Model	Most similar (1)	Most similar (2)	Most similar (3)	Most similar (4)	Most similar (5)
파일	10146	Word2vec	프로토콜	애플리케이션	url	디렉터리	포맷
		Fasttext	파일	확장자	음악파일	포멧	디렉터리
원인	11589	Word2vec	부작용	현상	요인	증상	질병
		Fasttext	주원인	초래	요인	직접	주요인
태생지	8	Word2vec	부타	구티에레즈	보아뱀	올림피코	집사장
		Fasttext	생지	탄생지	출생지	발생지	무생지
미스코리아	11	Word2vec	치평요람	神檀實記	컬투	방학기	김현승
		Fasttext	믹스코리아	라이코스코리아	악스코리아	보이스코리아	포브스코리아

- Document frequency (DF) 가 낮을 경우, **word2vec**은 유의미한 단어를 찾을 수 없음
- **Fasttext**의 경우, **subword information**이 유사한 단어를 찾았음

# fastText (Cont.)

## ■ 오토자, OOV, 등장 횟수가 적은 학습 단어에 대해 강세

Input word	FastText Model	Most similar words in range of top 3		
페널티 (Penalty) <b>OOV word</b>	Baseline	리날디 (Rinaldi)	페레티 (Ferretti)	마세티 (Machete)
	Jamo-advanced	페널티골 (Penalty goal)	페널티 (Penalty)	드록바 (Drogba)
나프탈렌 (Naphthalene) <b>OOV word</b>	Baseline	야렌 (Yaren)	콜루바라 (Kolubara)	몽클로아아라바카 (Moncloa-Aravaca)
	Jamo-advanced	나프탈렌 (Naphthalene)	테레프탈산 (Terephthalic acid)	아디프산 (Adipic acid)
스테이크 (Steak) <b>OOV word</b>	Baseline	스탠포드 (Stanhope)	스탠너드 (Stannard)	화이트스네이크 (White Snake)
	Jamo-advanced	롱테이크 (Long take)	비프스테이크 (Beefsteak)	스테이크 (Steak)

---

## 4. GloVe



# GloVe

## ■ GloVe(Global Vectors for Word Representation)

- Count 기반과 예측 기반 모두 사용하는 방법론
- 2014년에 Stanford Univ.에서 개발한 단어 Embedding 방법론.
- 기존의 카운트 기반의 LSA(Latent Semantic Analysis)와 예측 기반의 Word2Vec의 단점을 지적하며 이를 보완한다는 목적
- 실제로도 Word2Vec만큼이나 뛰어난 성능

# GloVe (Cont.)

## ■ LSA의 단점

- Count 기반
- Corpus의 전체적인 통계 정보를 고려하기는 하지만, 왕:남자 = 여왕:? (정답은 여자)와 같은 단어 의미의 유추 작업(Analogy task)에는 성능이 떨어짐.

## ■ Word2Vec의 단점

- 예측 기반
- 단어 간 유추 작업에는 LSA보다 뛰어나지만, Embedding Vector가 window 크기 내에서만 주변 단어를 고려
- Corpus의 전체적인 통계 정보를 반영하지 못함.

# Window based Co-occurrence Matrix

## ■ 단어의 동시 등장 행렬

- 행과 열을 전체 단어 집합의 단어들로 구성
- i 단어의 윈도우 크기(Window Size) 내에서 k 단어가 등장한 횟수를 i행 k열에 기재한 행렬

## ■ 아래와 같은 Text가 있을 때 :

- I like deep learning
- I like NLP
- I enjoy flying

카운트	I	like	enjoy	deep	learning	NLP	flying
I	0	2	1	0	0	0	0
like	2	0	0	1	0	1	0
enjoy	1	0	0	0	0	0	1
deep	0	1	0	0	1	0	0
learning	0	0	0	1	0	0	0
NLP	0	1	0	0	0	0	0
flying	0	0	1	0	0	0	0

# Co-occurrence Probability

## ■ 동시 등장 확률 $P(k | i)$

- Co-occurrence Matrix로부터 특정 단어  $i$ 의 전체 등장 횟수를 카운트하고
- 특정 단어  $i$ 가 등장했을 때 어떤 단어  $k$ 가 등장한 횟수를 카운트하여 계산한 조건부 확률

동시 등장 확률과 크기 관계 비(ratio)	$k=solid$	$k=gas$	$k=water$	$k=fasion$
$P(k   ice)$	0.00019	0.000066	0.003	0.000017
$P(k   steam)$	0.000022	0.00078	0.0022	0.000018
$P(k   ice) / P(k   steam)$	8.9	0.085	1.36	0.96

# Co-occurrence Probability (Cont.)

## ■ 동시 등장 확률 $P(k | i)$

- Co-occurrence Matrix로부터 특정 단어  $i$ 의 전체 등장 횟수를 카운트하고
- 특정 단어  $i$ 가 등장했을 때 어떤 단어  $k$ 가 등장한 횟수를 카운트하여 계산한 조건부 확률

단어의 동시 등장 확률과 크기 관계 비(ratio)	$k=solid$	$k=gas$	$k=water$	$k=fasion$
$P(k   ice)$	큰 값	작은 값	큰 값	작은 값
$P(k   steam)$	작은 값	큰 값	큰 값	작은 값
$P(k   ice) / P(k   steam)$	큰 값	작은 값	1에 가까움	1에 가까움

---

## 5. Pre-trained Word Embedding



# Keras Embedding Layer

## ■ Keras's **Embedding()**

- 단어들에 대해 Word Embedding을 수행하는 도구.
- Embedding Layer 구현.
- Embedding Layer를 사용하기 위해서 입력 Sequence의 각 입력은 모두 Integer Encoding이 되어야 한다.

■ 어떤 단어 → 단어에 부여된 고유한 정수값(Embedding Layer의 입력) → Embedding Layer 통과 → Dense Vector(Embedding Layer 출력)

■ Embedding Layer는 입력 정수에 대해 Dense Vector로 맵핑하고 이 Dense Vector는 ANN의 학습 과정에서 가중치가 학습되는 것과 같은 방식으로 훈련된다.

■ 이렇게 되면 단어는 결과적으로 Model이 해결하려는 문제에 특화된 Dense Vector가 된다.

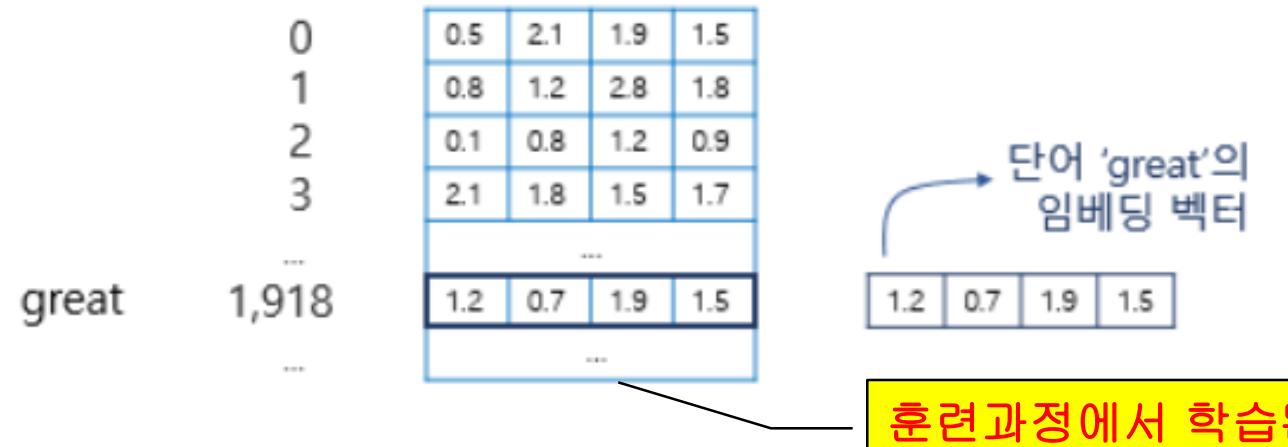
■ 이 Dense Vector를 **Embedding Vector**라고 부른다.

# Keras Embedding Layer (Cont.)

## ■ Embedding Layer를 지난다는 의미:

- 각 단어의 Embedding Vector값이 기록되어져 있다.
- Embedding Layer의 입력인 정수를 Index로 가지는 Table로부터 값을 가져오는 Lookup Table이라고 볼 수 있다.
- Table은 단어 집합의 크기만큼의 행을 가지므로 모든 단어는 고유한 Embedding Vector를 가진다.

Word → Integer → lookup Table → Embedding vector



# Keras Embedding Layer (Cont.)

- Keras의 Embedding Layer 구현 Code.

```
v = Embedding(20001, 256, input_length=500)
```

```
# vocab_size = 20001, output_dim = 256, input_length = 500
```

- 세 개의 인자

- **vocab\_size**

- Text Data의 전체 단어 집합의 크기.

- **output\_dim**

- Embedding 된 후의 단어의 차원

- **input\_length**

- 입력 시퀀스의 길이.

# Keras Embedding Layer (Cont.)

## ■ **Embedding ()**

- ① (number of samples, input\_length)인 2D 정수 Tensor를 입력 받는다.
- ② 이 때 각 sample은 Integer Encoding이 된 결과이다.
- ③ Integer Sequence이다.
- ④ Word Embedding 작업을 수행
- ⑤ (number of samples, input\_length, embedding word dimentinality)인 3D 실수 Tensor를 리턴한다.

# Keras Embedding Layer (Cont.)

## ■ Keras Embedding Layer Example

- 문장의 긍정, 부정을 판단하는 간단한 감성 분류 Model

```
1 sentences = ['멋있어 최고야 짱이야 감탄이다', '헛소리 지껄이네', '닥쳐 자식아', '#  
2           '우와 대단하다', '우수한 성적', '형편없다', '최상의 퀄리티']  
3 y_train = [1, 0, 0, 1, 1, 0, 1]
```

```
1 from keras.preprocessing.text import Tokenizer  
2 t = Tokenizer()  
3 t.fit_on_texts(sentences)  
4 vocab_size = len(t.word_index) + 1  
5  
6 print(vocab_size)
```

# Keras Embedding Layer (Cont.)

- 각 문장에 대해서 Integer Encoding 수행.

```
1 X_encoded = t.texts_to_sequences(sentences)
2 print(X_encoded)
```

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

```
1 max_len=max(len(l) for l in X_encoded)
2 print(max_len)
```

```
4
```

```
1 from keras.preprocessing.sequence import pad_sequences
2 X_train=pad_sequences(X_encoded, maxlen=max_len, padding='post')
3 print(X_train)
```

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

# Keras Embedding Layer (Cont.)

- 출력층에 1개의 Neuron에 Activation Function으로 Sigmoid Function를 사용하여 Binary Classification을 수행.

```
1 from keras.models import Sequential  
2 from keras.layers import Dense, Embedding, Flatten  
3  
4 model = Sequential()  
5 model.add(Embedding(vocab_size, 4, input_length=max_len)) # 모든 임베딩 벡터는 4차원을 가지게됨.  
6 model.add(Flatten()) # Dense의 입력으로 넣기위함.  
7 model.add(Dense(1, activation='sigmoid'))
```

```
1 model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])  
2 model.fit(X_train, y_train, epochs=100, verbose=2)
```

```
Epoch 18/100  
- 0s - loss: 0.6824 - acc: 0.8571  
Epoch 19/100  
- 0s - loss: 0.6810 - acc: 1.0000
```

# Pre-trained Word Embedding

## ■ Keras의 Embedding Layer

- Keras의 **Embedding()**
- 자연어 처리시 갖고 있는 훈련 데이터의 단어들을 Embedding Layer를 구현하여 Embedding Vector로 학습하는 경우

## ■ Pre-trained Word Embedding 사용하기

- Word2vec, FastText, Glove등을 통해서 이미 사전에 훈련된 Embedding Vector를 불러오는 방법

# Pre-Trained GloVe Embedding 사용하기

- 훈련 데이터가 적은 상황일 때
- 훈련 데이터가 적은 상황에서 Keras의 **Embedding()**으로 해당 문제에 충분히 특화된 Embedding Vector를 만들어내는 것이 어려움.
- Downloads : <http://nlp.stanford.edu/data/glove.6B.zip>

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

```
1 sentences = ['nice great best amazing', 'stop lies', 'pitiful nerd', '#  
2           'excellent work', 'supreme quality', 'bad', 'highly respectable']  
3 y_train = [1, 0, 0, 1, 1, 0, 1]  
4  
5 from keras.preprocessing.text import Tokenizer  
6 t = Tokenizer()  
7 t.fit_on_texts(sentences)  
8 vocab_size = len(t.word_index) + 1  
9  
10 X_encoded = t.texts_to_sequences(sentences)  
11 max_len=max(len(l) for l in X_encoded)  
12  
13 from keras.preprocessing.sequence import pad_sequences  
14 X_train=pad_sequences(X_encoded, maxlen=max_len, padding='post')  
15 print(X_train)
```

Using TensorFlow backend.

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

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

- \$ **unzip glove.6B.zip**
- 4개의 파일 중 사용할 파일은 glove.6B.100d.txt 파일이다.
- 해당 파일은 하나의 줄당 101개의 값을 가지는 리스트를 갖고 있다.
- 두 개의 줄만 읽어본다.

```
1 n = 0
2 f = open('./glove.6B.100d.txt', encoding="utf8")
3 for line in f:
4     word_vector = line.split() # 각 줄을 읽어와서 word_vector에 저장.
5     print(word_vector) # 각 줄을 출력
6     word = word_vector[0] # word_vector에서 첫번째 값만 저장
7     print(word) # word_vector의 첫번째 값만 출력
8     n=n+1
9     if n==2:
10         break
11 f.close()
```

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

```
[ 'the' , '-0.038194' , '-0.24487' , '0.72812' , '-0.39961' , '0.083172' , '0.043953' , '-0.39141' , '0.3344' ,  
'-0.57545' , '0.087459' , '0.28787' , '-0.06731' , '0.30906' , '-0.26384' , '-0.13231' , '-0.20757' , '0.3339'  
5' , '-0.33848' , '-0.31743' , '-0.48336' , '0.1464' , '-0.37304' , '0.34577' , '0.052041' , '0.44946' , '-0.4  
6971' , '0.02628' , '-0.54155' , '-0.15518' , '-0.14107' , '-0.039722' , '0.28277' , '0.14393' , '0.23464' ,  
'-0.31021' , '0.086173' , '0.20397' , '0.52624' , '0.17164' , '-0.082378' , '-0.71787' , '-0.41531' , '0.2033  
5' , '-0.12763' , '0.41367' , '0.55187' , '0.57908' , '-0.33477' , '-0.36559' , '-0.54857' , '-0.062892' , '0.  
26584' , '0.30205' , '0.99775' , '-0.80481' , '-3.0243' , '0.01254' , '-0.36942' , '2.2167' , '0.72201' , '-0.  
24978' , '0.92136' , '0.034514' , '0.46745' , '1.1079' , '-0.19358' , '-0.074575' , '0.23353' , '-0.052062' ,  
'-0.22044' , '0.057162' , '-0.15806' , '-0.30798' , '-0.41625' , '0.37972' , '0.15006' , '-0.53212' , '-0.205  
5' , '-1.2526' , '0.071624' , '0.70565' , '0.49744' , '-0.42063' , '0.26148' , '-1.538' , '-0.30223' , '-0.073  
438' , '-0.28312' , '0.37104' , '-0.25217' , '0.016215' , '-0.017099' , '-0.38984' , '0.87424' , '-0.72569' ,  
'-0.51058' , '-0.52028' , '-0.1459' , '0.8278' , '0.27062']
```

the

```
[ 'the' , '-0.10767' , '0.11053' , '0.59812' , '-0.54361' , '0.67396' , '0.10663' , '0.038867' , '0.35481' , '0.0  
6351' , '-0.094189' , '0.15786' , '-0.81665' , '0.14172' , '0.21939' , '0.58505' , '-0.52158' , '0.22783' , '-  
0.16642' , '-0.68228' , '0.3587' , '0.42568' , '0.19021' , '0.91963' , '0.57555' , '0.46185' , '0.42363' , '-  
0.095399' , '-0.42749' , '-0.16567' , '-0.056842' , '-0.29595' , '0.26037' , '-0.26606' , '-0.070404' , '-0.2  
7662' , '0.15821' , '0.69825' , '0.43081' , '0.27952' , '-0.45437' , '-0.33801' , '-0.58184' , '0.22364' , '-  
0.5778' , '-0.26862' , '-0.20425' , '0.56394' , '-0.58524' , '-0.14365' , '-0.64218' , '0.0054697' , '-0.3524  
8' , '0.16162' , '1.1796' , '-0.47674' , '-2.7553' , '-0.1321' , '-0.047729' , '1.0655' , '1.1034' , '-0.220  
8' , '0.18669' , '0.13177' , '0.15117' , '0.7131' , '-0.35215' , '0.91348' , '0.61783' , '0.70992' , '0.2395  
5' , '-0.14571' , '-0.37859' , '-0.045959' , '-0.47368' , '0.2385' , '0.20536' , '-0.18996' , '0.32507' , '-1.  
1112' , '-0.36341' , '0.98679' , '-0.084776' , '-0.54008' , '0.11726' , '-1.0194' , '-0.24424' , '0.12771' ,  
'0.013884' , '0.080374' , '-0.35414' , '0.34951' , '-0.7226' , '0.37549' , '0.4441' , '-0.99059' , '0.61214' ,  
'-0.35111' , '-0.83155' , '0.45293' , '0.082577']
```

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

```
1 print(type(word_vector))
2 print(len(word_vector))
```

```
<class 'list'>
101
```

- 101개의 값 중에서 첫 번째 값은 Embedding Vector가 의미하는 단어를 의미하며, 두 번째 값부터 마지막 값은 해당 단어의 Embedding Vector의 100개의 차원에서의 각 값을 의미
- 즉, glove.6B.100d.txt는 수많은 단어에 대해서 100개의 차원을 가지는 Embedding Vector로 제공하고 있다.
- 위의 출력 결과는 단어 *the*에 대해서 100개의 차원을 가지는 Embedding Vector와 단어 *,*에 대해서 100개의 차원을 가지는 Embedding Vector이다.

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

- glove.6B.100d.txt에 있는 모든 Embedding Vector들을 불러온다.
- 형식은 키(key)와 값(value)의 쌍(pair)를 가지는 Python's dict구조이다.

```
1 import numpy as np
2 embedding_dict = dict()
3 f = open('./glove.6B.100d.txt', encoding="utf8")
4
5 for line in f:
6     word_vector = line.split()
7     word = word_vector[0]
8     word_vector_arr = np.asarray(word_vector[1:], dtype='float32')
9     # 100개의 값을 가지는 array로 변환
10    embedding_dict[word] = word_vector_arr
11 f.close()
12 print('%s개의 Embedding vector가 있습니다.' % len(embedding_dict))
```

400000개의 Embedding vector가 있습니다.

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

- 임의의 단어 *respectable*에 대해서 Embedding Vector를 출력해본다.

```
1 print(embedding_dict['respectable'])
2 print(len(embedding_dict['respectable']))
```

```
[-0.049773  0.19903   0.10585   0.1391   -0.32395   0.44053
 0.3947   -0.22805  -0.25793   0.49768   0.15384  -0.08831
 0.0782   -0.8299   -0.037788  0.16772  -0.45197  -0.17085
 0.74756   0.98256   0.81872   0.28507   0.16178  -0.48626
-0.006265  -0.92469  -0.30625  -0.067318  -0.046762  -0.76291
-0.0025264 -0.018795  0.12882  -0.52457   0.3586   0.43119
-0.89477   -0.057421  -0.53724   0.25587   0.55195   0.44698
-0.24252   0.29946   0.25776  -0.8717   0.68426  -0.05688
-0.1848    -0.59352  -0.11227  -0.57692  -0.013593  0.18488
-0.32507   -0.90171   0.17672   0.075601  0.54896  -0.21488
-0.54018   -0.45882  -0.79536   0.26331  0.18879  -0.16363
 0.3975    0.1099   0.1164   -0.083499  0.50159   0.35802
 0.25677   0.088546  0.42108   0.28674  -0.71285  -0.82915
 0.15297   -0.82712  0.022112  1.067   -0.31776  0.1211
-0.069755  -0.61327  0.27308  -0.42638  -0.085084  -0.17694
-0.0090944 0.1109   0.62543  -0.23682  -0.44928  -0.3667
-0.21616   -0.19187  -0.032502  0.38025 ]
100
```

- Vector값이 출력되며 길이는 100인 것을 확인할 수 있다.

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

- 훈련 Data의 단어 집합의 모든 단어에 대해서 사전 훈련된 GloVe의 Embedding Vector들을 맵핑한다.

```
1 embedding_matrix = np.zeros((vocab_size, 100))
2 # 단어 집합 크기의 행과 100개의 열을 가지는 행렬 생성. 값은 전부 0으로 채워진다.
3 np.shape(embedding_matrix)
```

(16, 100)

```
1 print(t.word_index.items())
```

```
dict_items([('nice', 1), ('great', 2), ('best', 3), ('amazing', 4), ('stop', 5), ('lies', 6), ('pitiful', 7), ('nerd', 8), ('excellent', 9), ('work', 10), ('supreme', 11), ('quality', 12), ('bad', 13), ('highly', 14), ('respectable', 15)])
```

```
1 for word, i in t.word_index.items():
2     # 훈련 데이터의 단어 집합에서 단어를 1개씩 꺼내온다.
3     temp = embedding_dict.get(word)
4     # 단어(key) 해당되는 임베딩 벡터의 100개의 값(value)를 임시 변수에 저장
5     if temp is not None:
6         embedding_matrix[i] = temp
7         # 임수 변수의 값을 단어와 맵핑되는 인덱스의 행에 삽입
```

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

- GloVe의 Embedding Vector를 이용하여 Embedding Layer를 만든다.

```
1 from keras.models import Sequential  
2 from keras.layers import Dense, Embedding, Flatten  
3  
4 model = Sequential()  
5 e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=max_len, trainable=False)  
6
```

- 현재 실습에서 사전 훈련된 Word Embedding을 100차원의 값인 것으로 사용하고 있기 때문에 Embedding Layer의 **output\_dim**의 인자값으로 100을 주어야 한다.
- 사전 훈련된 Word Embedding을 그대로 사용할 것이므로, 별도로 더 이상 훈련을 하지 않는다는 옵션을 준다.
- 이렇게 하기 위해 **trainable=False**로 선택할 수 있다.

# Pre-Trained GloVe Embedding 사용하기 (Cont.)

```
1 model.add(e)
2 model.add(Flatten())
3 model.add(Dense(1, activation='sigmoid'))
```

```
1 model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
2 model.fit(X_train, y_train, epochs=100, verbose=2)
```

```
Epoch 96/100
- 0s - loss: 0.1126 - acc: 1.0000
Epoch 97/100
- 0s - loss: 0.1111 - acc: 1.0000
Epoch 98/100
- 0s - loss: 0.1097 - acc: 1.0000
Epoch 99/100
- 0s - loss: 0.1083 - acc: 1.0000
Epoch 100/100
- 0s - loss: 0.1069 - acc: 1.0000
```

```
<keras.callbacks.History at 0x7fb66c59160>
```

# Word Embedding 한계점

- Word2Vec이나 fastText와 같은 Word Embedding 방식은 동형어, 다의어 등에 대해서는 Embedding 성능이 좋지 못하다는 단점
- 주변 단어를 통해 학습이 이루어지기 때문에 **문맥**을 고려할 수 없음.

## Account

- 1) I **account** him to be my friend ~라고 생각하다
- 2) He is angry on **account** of being excluded from the invitation 이유, 근거
- 3) I have an **account** with the First National Bank ~때문에
- 4) The police wrote an **account** of the accident 보고서
- 5) Your check has been properly credited, and your **account** is now full? 계좌



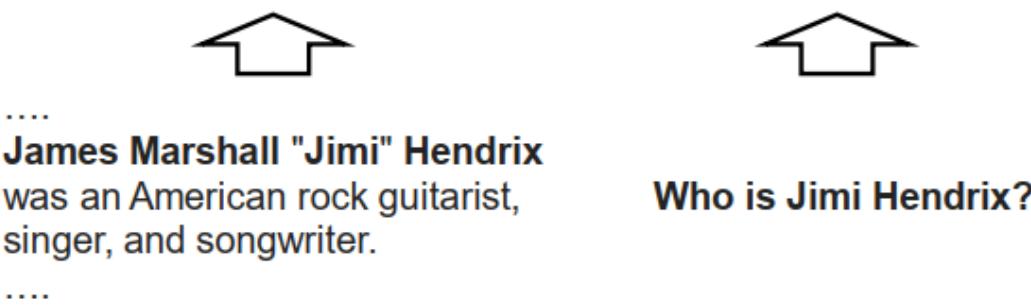
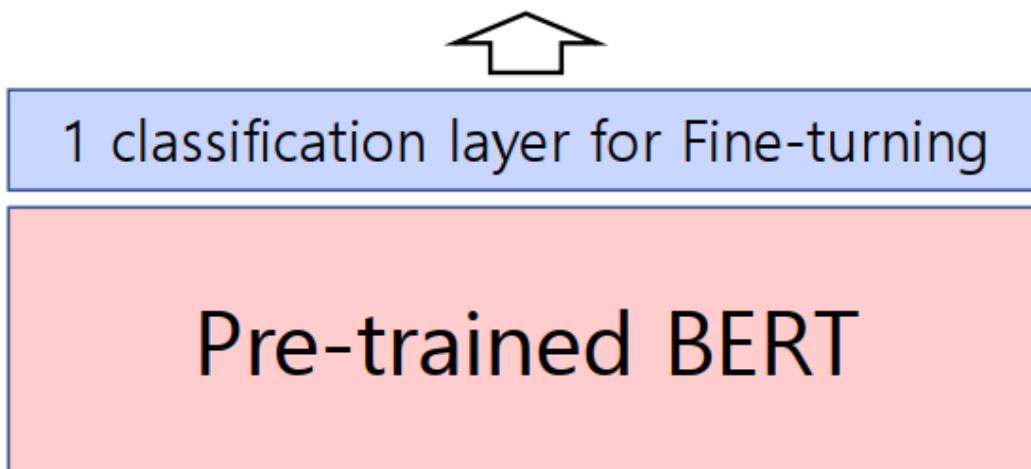
## 6. BERT

# BERT is...

- BERT는 bi-directional Transformer로 이루어진 언어모델
- 잘 만들어진 BERT 언어모델 위에 1개의 classification layer만 부착하여 다양한 NLP task를 수행
- 영어권에서 11개의 NLP task에 대해 state-of-the-art (SOTA) 달성

"Jimi" Hendrix was an American rock guitarist, singer, and songwriter.

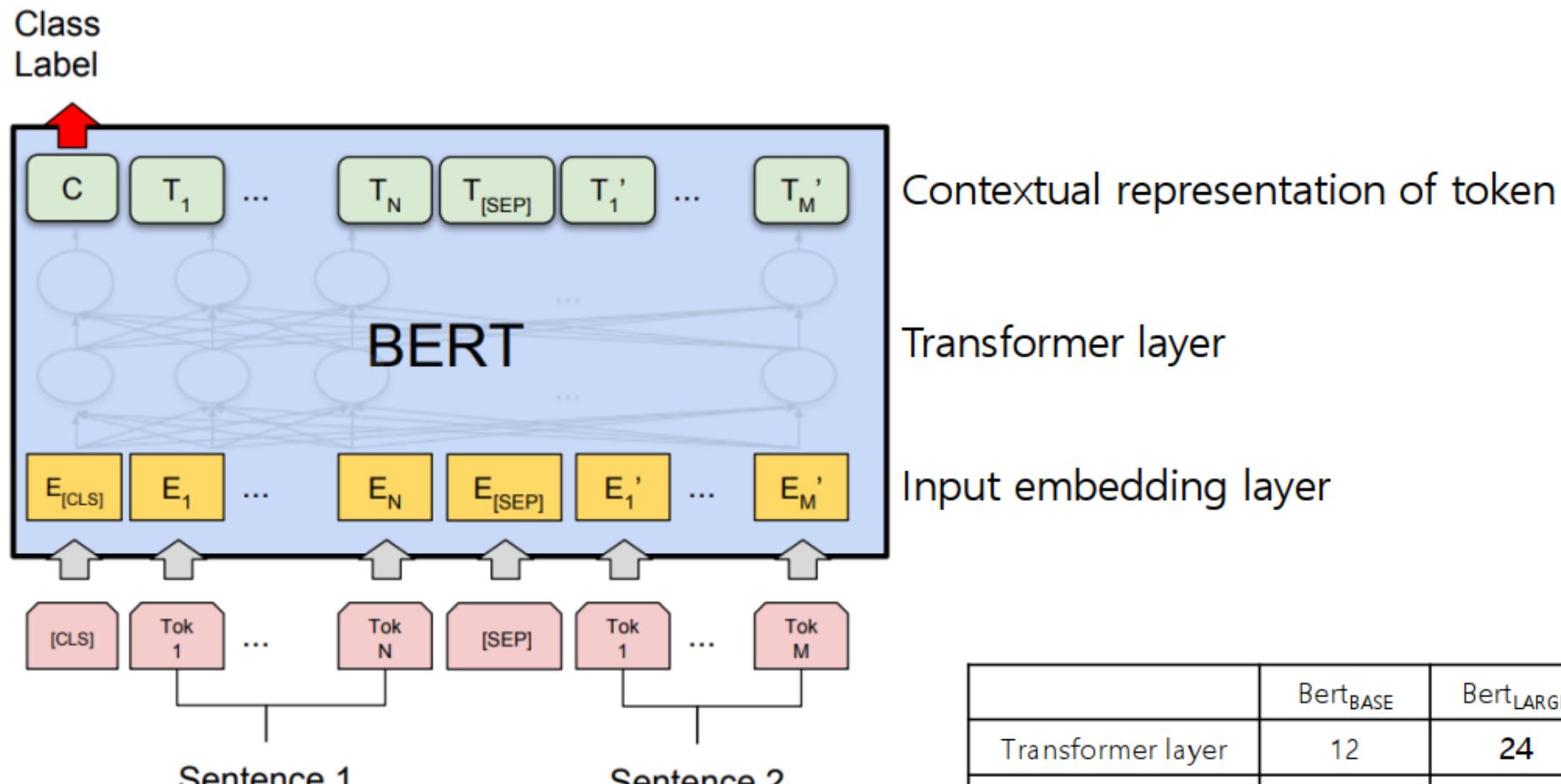
SQuAD v1.1 dataset leaderboard



Rank	Model	EM	F1
	Human Performance Stanford University (Rajpurkar et al. '16)	82.304	91.221
1	BERT (ensemble) Google AI Language <a href="https://arxiv.org/abs/1810.04805">https://arxiv.org/abs/1810.04805</a>	87.433	93.160
2	BERT (single model) Google AI Language <a href="https://arxiv.org/abs/1810.04805">https://arxiv.org/abs/1810.04805</a>	85.083	91.835

# BERT Architecture

- Model architecture

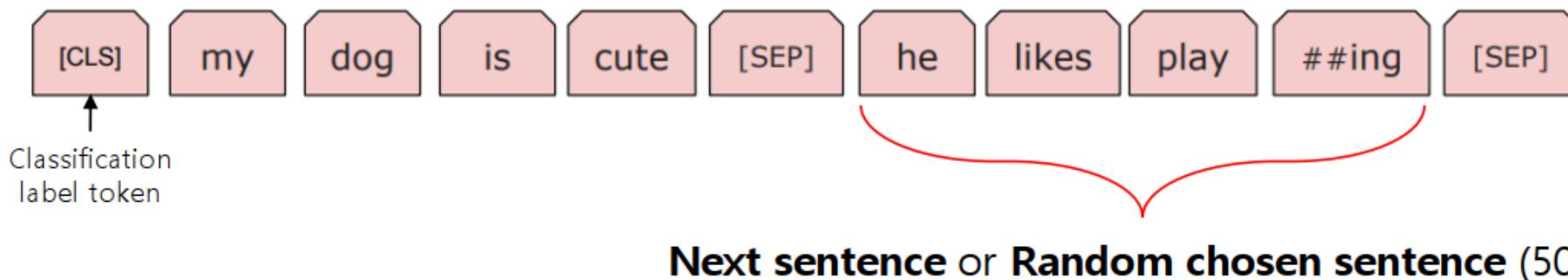


	Bert <sub>BASE</sub>	Bert <sub>LARGE</sub>
Transformer layer	12	24
Self-attention head	12	16
Total	110M	340M

# BERT Model Learning Data

- 학습 코퍼스 데이터
  - BooksCorpus (800M words)
  - English Wikipedia (2,500M words without lists, tables and headers)
  - 30,000 token vocabulary
- 데이터의 tokenizing
  - **WordPiece** tokenizing
    - He likes playing → He likes play ##ing
  - 입력 문장을 tokenizing하고, 그 token들로 'token sequence'를 만들어 학습에 사용
  - 2개의 token sequence가 학습에 사용

Example of two sentences token sequence



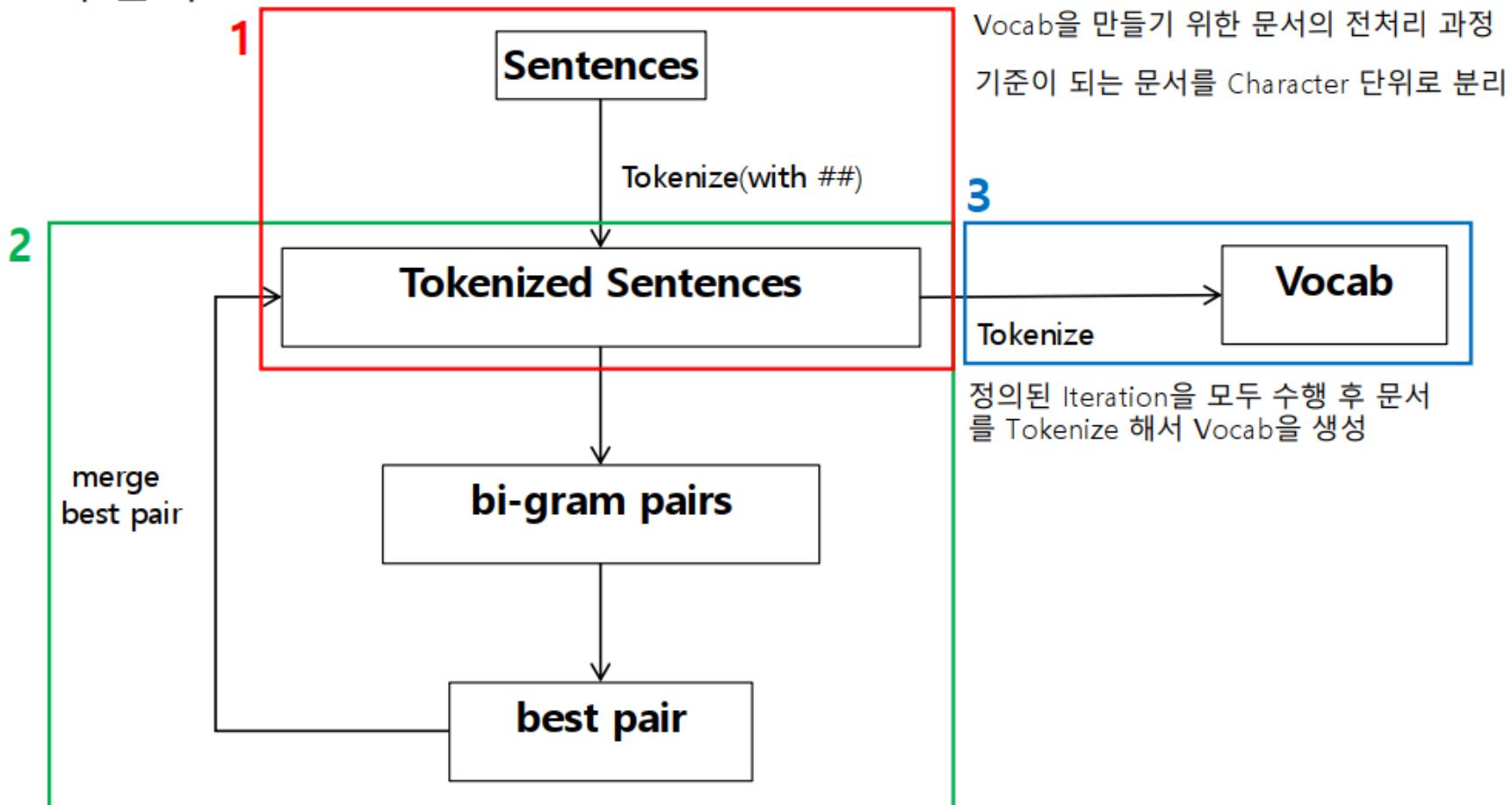
# BERT WordPiece Tokenizing

- Byte Pair Encoding (BPE) 알고리즘 이용
- 빈도수에 기반해 단어를 의미 있는 패턴(Subword)으로 잘라서 tokenizing



# BERT WordPiece Tokenizing (Cont.)

- BPE의 순서도



Character 단위로 분리된 문서에서 가장 많이 등장하는 Bi-gram Pair를 찾고 합쳐주는 과정

Iteration을 정해 놓고 주어진 횟수 만큼 수행

Vocab을 만들기 위한 문서의 전처리 과정  
기준이 되는 문서를 Character 단위로 분리

정의된 Iteration을 모두 수행 후 문서를 Tokenize 해서 Vocab을 생성

# BERT WordPiece Tokenizing (Cont.)

- Tokenize

경찰청 철창살은 외철창살이고 검찰청 철창살은 쌍철창살이다

Sentence		Tokenized Sentence (iter=0)
경찰청	→경 찰 청	→ 경 ##찰 ##청
철창살은	→철 창 살 은	→ 철 ##창 ##살 ##은
외철창살이고	→외 철 창 살 이 고	→ 외 ##철 ##창 ##살 ##이 ##고
검찰청	→검 찰 청	→ 검 ##찰 ##청
철창살은	→철 창 살 은	→ 철 ##창 ##살 ##은
쌍철창살이다	→쌍 철 창 살 이 다	→ 쌍 ##철 ##창 ##살 ##이 ##다

- 같은 글자라고 맨 앞에 나오는 것과 아닌 것에는 차이가 있다고 가정
- BERT의 경우 뒷단어에 '##'을 붙여서 구별 e.g.) '철' ≠ '##철' , '철창살' ≠ '##철창살'

# BERT WordPiece Tokenizing (Cont.)

- Building Vocab

Tokenized Sentence (iter=0)	Vocab (iter=0) <b>Vocab 후보!</b>
경 ##찰 ##청	경 ##찰 ##청
철 ##창 ##살 ##은	철 ##창 ##살 ##은
외 ##철 ##창 ##살 ##이 ##고	외 ##철 ##이 ##고
검 ##찰 ##청	검
철 ##창 ##살 ##은	
쌍 ##철 ##창 ##살 ##이 ##다	쌍 ##다

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해서 생성

# BERT WordPiece Tokenizing (Cont.)

- Bi-gram Pair Count

Vocab 후보 기준으로

Tokenized Sentence (iter=0)

경 ##찰 ##청

철 ##창 ##살 ##은

외 ##철 ##창 ##살 ##이 ##고

검 ##찰 ##청

철 ##창 ##살 ##은

쌍 ##철 ##창 ##살 ##이 ##다

Bi-gram pairs (iter=1)

→ (경, ##찰), (##찰, ##청)

→ (철, ##창), (##창, ##살), (##살, ##은)

→ (외 ##철), (##철, ##창), (##창, ##살),  
(##살, ##이), (##이, ##고)

→ (검, ##찰), (##찰, ##청)

→ (철, ##창), (##창, ##살), (##살, ##은),

→ (쌍 ##철), (##철, ##창), (##창, ##살),  
(##살, ##이), (##이, ##다)

(경, ##찰) :1    (##살, ##은) :2    (##이, ##고) :1

(##찰, ##청) :2    (외 ##철) :1    (검, ##찰) :1

(철, ##창) :2    (##철, ##창) :2    (쌍 ##철) :1

(##창, ##살) :4    (##살, ##이) :2    (##이, ##다) :1

# BERT WordPiece Tokenizing (Cont.)

- Merge Best pair

Best Pair: ##창 ##살 → **##창살**

Tokenized Sentence (iter=0)

경 ##찰 ##청

철 ##창 ##살 ##은

외 ##철 ##창 ##살 ##이 ##고

검 ##찰 ##청

철 ##창 ##살 ##은

쌍 ##철 ##창 ##살 ##이 ##다

Tokenized Sentence (iter=1)

→ 경 ##찰 ##청

→ 철 **##창살** ##은

→ 외 ##철 **##창살** ##이 ##고

→ 검 ##찰 ##청

→ 철 **##창살** ##은

→ 쌍 ##철 **##창살** ##이 ##다

# BERT WordPiece Tokenizing (Cont.)

- Building Vocab

Tokenized Sentence (iter=1)			Vocab 후보 업데이트
			Vocab (iter=1)
경	##찰	##청	경
철	##창살	##은	철
외	##철	##창살	##이
검	##찰	##청	##고
철	##창살	##은	외
쌍	##철	##창살	##철
	##이	##다	검
	##다		쌍

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해서 생성

# BERT WordPiece Tokenizing (Cont.)

- Bi-gram Pair Count

## Vocab 후보 기준으로

Tokenized Sentence (iter=1)

경 ##찰 ##청

철 ##창살 ##은

외 ##철 ##창살 ##이 ##고

검 ##찰 ##청

철 ##창살 ##은

쌍 ##철 ##창살 ##이 ##다

Bi-gram pairs (iter=2)

→ (경, ##찰), (##찰, ##청)

→ (철, ##창살), (##창살, ##은)

→ (외 ##철), (##철, ##창살), (##창살, ##이), (##이, ##고)

→ (검, ##찰), (##찰, ##청)

→ (철, ##창살), (##창살, ##은),

→ (쌍 ##철), (##철, ##창살), (##창살, ##이), (##이, ##다)

(경, ##찰) :1 (외 ##철) :1 (검, ##찰) :1

(##찰, ##청) :2 (##철, ##창살) :2 (쌍 ##철) :1

(철, ##창살) :2 (##창살, ##이) :2 (##이, ##다) :1

(##창살, ##은) :2 (##이, ##고) :1

# BERT WordPiece Tokenizing (Cont.)

- Merge Best pair

Best Pair: ##찰 ##청 → ##찰청

Tokenized Sentence (iter=1)

경 ##찰 ##청

철 ##창살 ##은

외 ##철 ##창살 ##이 ##고

검 ##찰 ##청

철 ##창살 ##은

쌍 ##철 ##창살 ##이 ##다

Tokenized Sentence (iter=2)

→ 경 ##찰청

→ 철 ##창살 ##은

→ 외 ##철 ##창살 ##이 ##고

→ 검 ##찰청

→ 철 ##창살 ##은

→ 쌍 ##철 ##창살 ##이 ##다

- Best pair가 여러 개여도 하나만 선택

# BERT WordPiece Tokenizing (Cont.)

- Building Vocab

Tokenized Sentence (iter=2)	Vocab 후보 업데이트
경 ##찰청	경 ##찰청
철 ##창살 ##은	철 ##창살 ##은
외 ##철 ##창살 ##이 ##고	외 ##철 ##이 ##고
검 ##찰청	검
철 ##창살 ##은	
쌍 ##철 ##창살 ##이 ##다	쌍 ##다

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해서 생성

# BERT WordPiece Tokenizing (Cont.)

- Bi-gram Pair Count

Vocab 후보 기준으로

Tokenized Sentence (iter=2)	Bi-gram pairs (iter=3)
경 ##찰청	→ (경, ##찰청)
철 ##창살 ##은	→ (철, ##창살), (##창살, ##은)
외 ##철 ##창살 ##이 ##고	→ (외 ##철), (##철, ##창살), (##창살, ##이), (##이, ##고)
검 ##찰청	→ (검, ##찰청)
철 ##창살 ##은	→ (철, ##창살), (##창살, ##은),
쌍 ##철 ##창살 ##이 ##다	→ (쌍 ##철), (##철, ##창살), (##창살, ##이), (##이, ##다)

(경, ##찰청)	:1	(##철, ##창살)	:2	(쌍 ##철)	:1
(철, ##창살)	:2	(##창살, ##이)	:2	(##이, ##다)	:1
(##창살, ##은)	:2	(##이, ##고)	:1		
(외 ##철)	:1	(검, ##찰청)	:1		

# BERT WordPiece Tokenizing (Cont.)

- Merge Best pair

Best Pair: 철 ##창살 → **철창살**

Tokenized Sentence (iter=2)

경 ##찰 ##청

철 ##창살 ##은

외 ##철 ##창살 ##이 ##고

검 ##찰 ##청

철 ##창살 ##은

쌍 ##철 ##창살 ##이 ##다

Tokenized Sentence (iter=3)

→ 경 ##찰청

→ **철창살** ##은

→ 외 ##철 ##창살 ##이 ##고

→ 검 ##찰청

→ **철창살** ##은

→ 쌍 ##철 ##창살 ##이 ##다

- 3 글자 이상의 패턴도 Iteration이 진행되면서 나타날 수 있음

# BERT WordPiece Tokenizing (Cont.)

- Building Vocab

Tokenized Sentence (iter=3)	Vocab (iter=3) <b>Vocab에 저장!</b>
경 ##찰청	경 ##찰 청
철창살 ##은	철창살 ##은
외 ##철 ##창살 ##이 ##고	외 ##철 ##창 살 ##이 ##고
검 ##찰청	검
철창살 ##은	
쌍 ##철 ##창살 ##이 ##다	쌍 ##다

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해서 생성(매 iteration마다 Vocab을 생성하는 것이 아님)

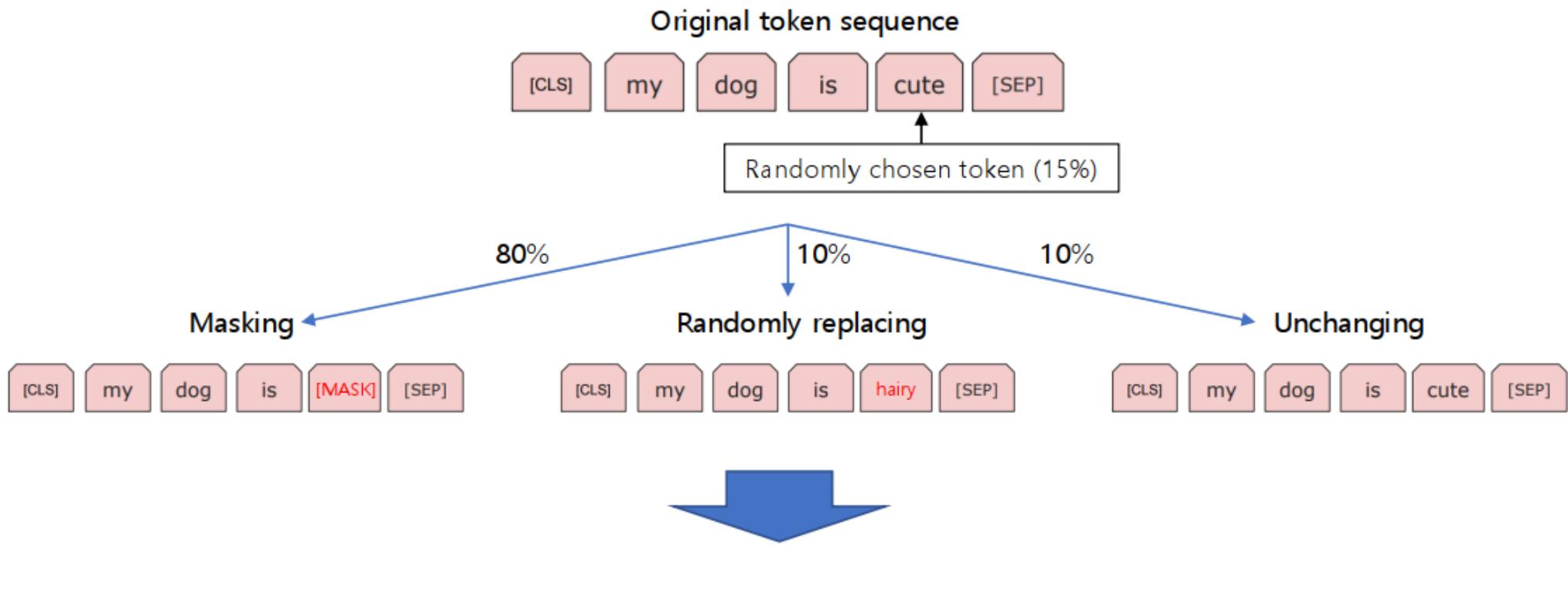
# BERT WordPiece Tokenizing (Cont.)

- BERT tokenization 예시
  - BPE로 subword로 분리했음에도 불구하고 vocab에 존재하지 않는 단어는 [UNK] 토큰으로 변환

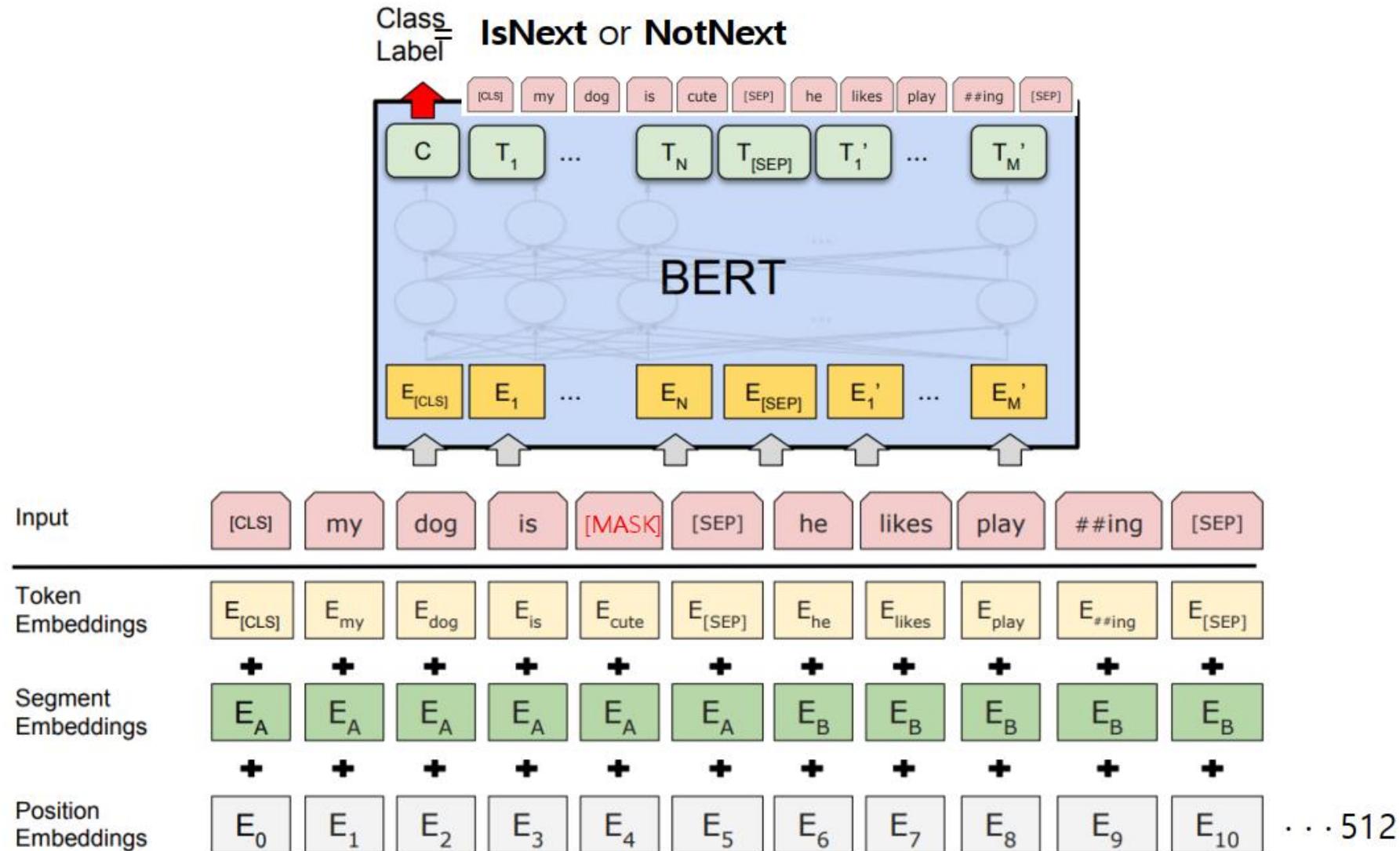
```
[INFO:tensorflow:tokens: [CLS] 유스티니아 ##누스의 분노 ##는 찌 ##를 듯 ##했으나 수도 ##를 떠나는 실수를 한 바람에 팔리 ##피 ##> 쿠스가 콘스탄티노폴리스 ##를 차지하였고 유스티니아 ##누스는 11월 4일 불잡혀 처형 ##당했다 ##. [SEP] ##여 ##비 ##렸고 유스티니아 ##누스의 아들 6 [MASK] 티 ##베리우스 ##도 블라 ##케 ##르나이 성당에서 처형 ##당 ##했는데 이로써 [MASK] ##클리우스 왕조의 혈 ##통 ##은 완전히 끊 ##어졌다 ##. [UNK] 전동차의 공기 ##제동 방식 가운데 하나이다 ##. 1868년 ##에 조지 웨 ##스팅 [MASK] 발명 ##하 > 였다 ##. 순수 ##하게 공기 ##압 제어로 동작 ##하는 것은 크게 단행 ##운전용인 [UNK] 연결 ##운전용인 비상 ##변 부착 [UNK] 나뉜다 # #. 개요 SM 공급 [UNK] Reser ##v ##o ##ir ##: [UNK] 불리는 가 ##압 ##된 공기 탱 ##크로부터 ##, 운전 ##대 ##까지 연결된 공기 ##저 장관으로 불리는 공기관 ##을 통해 공기 ##압 ##을 공급 ##하여 제동 ##변을 조작 ##해 개폐 ##하는 것으로 ##, [UNK] Air P ##ip ##e # #: [UNK] 불리는 브레이크 실린 ##더 직결 공기관 ##에 가 ##압 ##하여 제동력을 얻는 매우 단순한 제동 시스템이다 ##. 웨 ##스팅 ##하 우스가 붙인 공식 이름은 [UNK] a ##ir b ##ra ##ke ##/ ##M ##otor c ##ar ##: 전동차 641 [UNK] 세계 여러 나라의 단행 ##운전차량에 널리 보급 [MASK] 노 ##면 ##전차 ##의 경우 아직도 영업 일 ##선에서 많이 쓰이고 있다. 다만 이 시스템은 구조가 간단 ##하며 동작 ##이 신 ##속 ##하고 확실 ##하지만 ##, 공기관 ##이 파 ##순 ##될 경우 제동 ##이 걸 ##리지 않게 되는 위험 ##이 있기 때문에 ##, 보안 ##상 연결 운전에는 사용할 수 없다는 단 ##점이 있다. SME SM [MASK] ##에서 문제가 된 열차 ##분리 [MASK] 발생 등의 대응 ##책 ##으 > 로 비상용 자동 ##공기 ##제동 [UNK] 그 지 ##령에 이용하는 [UNK] [UNK] 함께 설치한 [UNK] a ##ir b ##ra ##ke ##/ ##M ##otor c ##ar ##/ ##E ##mer ##gen [MASK] v ##al ##ve ##: 전동차 ##용 비상 ##변 부착 직통 ##공기 ##제동 ##. 모터 ##가 없는 트 ##레일 ##러 ##> 용 ##은 ST ##E 혹은 [UNK] 웨 ##스팅 ##하우스 ##사가 개발 ##하여 2~ ##3 ##량 정도의 단편성 ##용으로 보급 ##되었다 ##. 이 SME ##> 는 원형 ##인 SM ##과 같은 직통제동기구를 이루고 있지만 ##, [MASK] 저장기에 [MASK] 공기 ##탱 ##크가 원 ##공기 [UNK] Reser [MASK] ##o ##ir ##: [UNK] 불리며 공기 저장관 ##도 원 ##공기 [UNK] [MASK] ##v ##o ##ir P ##ip ##e ##: [UNK] 불리고 있다. 이것은 SME [MASK] [MASK] ##제동부에 비상 ##제동의 동력 ##원을 공급 ##하는 보조 공기 ##저장기가 존재 ##하기 때문에 ##, [MASK] 구별 ##하기 위 ##함이다 ##. 비상 ##변 ##에는 평 ##상 ##시는 490 ##k ##P ##a ##의 압 ##력이 가해 ##지고 있어 ##, 긴급 ##할 때 ##나 비상 ##변 ##의 호 ##스가 파 ##열 ##되었을 때도 비상 ##제동 ##이 작동 ##한다 ##. 비상 ##제동 ##은 자동 ##공기 ##제동 ##과 같이 보조 공기 ##> 저장기의 공기를 배출시키 ##는 것으로 작동시키기 때문에 ##, 안전성이 향상 ##되었다 ##. 브레이크의 가 ##감 ##압 ##은 [MASK] ##래 > 의 SM 방식 ##과 달리 가 ##감 ##압 ##의 속도가 언제나 정해져 있다. 제동 단계는 [UNK] [UNK] [UNK] [UNK] 압 ##력을 [UNK] [UNK] 압 ##력을 [UNK] 이렇게 4 ##가지 ##가 있다. 제2차 십자군 ( ##114 ##7년 - [UNK] 제1차 십자군 [MASK] 이후 팔레스타 ##인의 십자 >
```

# BERT Masking 기법

- 데이터의 tokenizing
  - Masked language model (MLM): input token을 일정 확률로 masking



# BERT Masking 기법 (Cont.)



---

## **II. Text Classification**



---

# 1. Text Classification using Keras



# 훈련 데이터에 대한 이해

- Naïve Bayes Classification를 통한 Text 분류
- Supervised Learning.
- Label(정답 Data)이 필요.

텍스트(메일의 내용)	레이블(스팸 여부)
당신에게 드리는 마지막 혜택! ...	스팸 메일
내일 볼 수 있을지 확인 부탁...	정상 메일
쉿! 혼자 보세요...	스팸 메일
언제까지 답장 가능할...	정상 메일
...	...
(광고) 멋있어질 수 있는...	스팸 메일

# 훈련 데이터에 대한 이해 (Cont.)

## ■ 예) Spam Mail Classification

- 훈련 데이터
  - Mail의 내용
  - Label 필요(해당 Mail이 Spam인지 여부)
- 기계는 Data가 잘 설계되었다면 훈련 Data에 없는 어떤 Mail Text가 주어졌을 때 Label을 예측하게 됨.



# 훈련 데이터와 테스트 데이터

- 갖고 있는 Mail Data Sample 모두를 훈련에 사용하기 보다 Test용 Data를 갖고 있는 것이 좋다.
- 만일 Sample Data가 20,000개이면, 18,000개의 Sample은 훈련용으로 사용하고, 2,000개의 Sample은 Test용으로 보류한다.
- 18,000개의 Sample로 Model이 훈련이 다 되면, 2,000개의 Test Sample로 정확도를 확인할 수 있다.
- 예제에서는 갖고 있는 데이터에서 분류하고자 하는 Text Data의 열을 **x**, Label Data의 열을 **y**라고 명명한다.
- 이것을 훈련 데이터(**x\_train**, **y\_train**)와 테스트 데이터(**x\_test**, **y\_test**)로 분리한다.
- Model은 **x\_train**과 **y\_train**을 학습하고, **x\_test**에 대해서 Label을 예측한다.
- 그리고 Model이 예측한 Label과 **y\_test**를 비교해서 정답율을 계산한다.

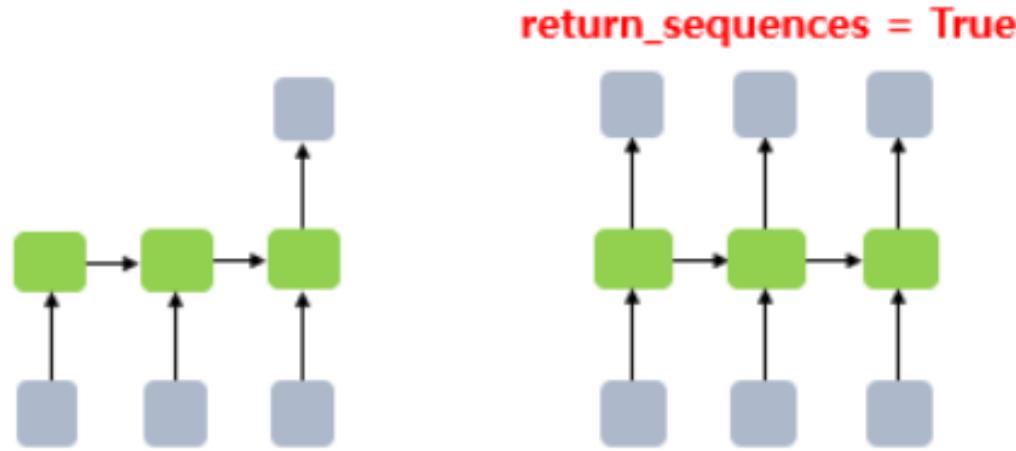
## 단어에 대한 Index 부여

- Keras의 Embedding()은 단어 각각에 대해 정수가 Mapping된 입력에 대해서 Embedding 작업 수행.
- 단어 각각에 숫자 Mapping, Index를 부여하는 방법
  - Integer Encoding과 같이 단어를 빈도수 순대로 정렬하고 순차적으로 Index를 부여
- 장점 :
  - 등장 빈도수가 적은 단어의 제거
  - 등장 빈도수 내림차순일 때 뒤로 갈 수록 빈도수가 낮아짐으로 상대적으로 앞에 있는 단어만 처리 가능.

# RNN으로 분류하기

- `model.add(SimpleRNN(hidden_size, input_shape=(timesteps, input_dim)))`
- LSTM, GRU도 Parameter 동일
- Parameters
  - **hidden\_size**
    - 출력의 크기(**output\_dim**).
  - **timesteps**
    - 시점의 수 = 각 문서에서의 단어 수.
  - **input\_dim**
    - 입력의 크기 = 각 단어의 Vector 표현의 차원 수.

# RNN의 Many-to-One 문제



- Text Classification은 RNN의 다-대-일(Many-to-One) 문제에 속한다.
- 모든 시점(time-step)에 대해서 입력을 받지만 최종 시점의 RNN Cell만이 은닉 상태를 출력하고, 이것이 출력층으로 가서 활성화 함수를 통해 정답을 고르는 문제가 된다.
- 2개의 선택지 중 정답 고르기 → Binary Classification 문제
- 3개 이상의 선택지 중에서 정답 고르기 → Multi-Class Classification 문제

# RNN의 Many-to-One 문제 (Cont.)

- 각각 문제에 맞는 다른 활성화 함수와 손실 함수를 사용한다.
- Binary Classification 문제
  - 출력층의 활성화 함수로 Sigmoid 함수 사용
  - 손실 함수로 **binary\_crossentropy** 사용
  - Spam Mail Classification
  - IMDB Review Sentiment Classification
- Multi-Class Classification 문제
  - 출력층의 활성화 함수로 **Softmax** 함수
  - 손실 함수로 **categorical\_crossentropy** 사용
  - Class가 N개라면 출력층에 해당되는 밀집층(dense layer)의 크기는 N이어야 한다.
  - Reuters News Classification

---

## 2. Spam Detection



# Spam Detection

## ■ 개요

- Data
  - Kaggle에서 제공
  - 정상 Mail과 Spam Mail이 섞여져 있는 Spam Mail Data
- 과정
  - Data에 대한 전처리를 진행
  - 바닐라 RNN(Vanilla RNN)을 이용한 Spam Mail Classification 구현

# Spam Mail Data에 대한 이해

## ■ Data Download :

- <https://www.kaggle.com/uciml/sms-spam-collection-dataset>

The screenshot shows the Kaggle homepage with a navigation bar at the top. Below the navigation bar, a banner for the "SMS Spam Collection Dataset" is displayed. The banner features a background image of a person's hand holding a smartphone. The dataset title "SMS Spam Collection Dataset" is prominently shown, along with the subtitle "Collection of SMS messages tagged as spam or legitimate". A "UCI Machine Learning" badge indicates the source, and a note states it was updated 3 years ago (Version 1). Below the banner, there are tabs for "Data", "Kernels (317)", "Discussion (3)", "Activity", and "Metadata". The "Data" tab is currently selected. To the right of these tabs are links for "Download (492 KB)" and "New Notebook". At the bottom of the banner, there are sections for "Usability" (rating 5.9) and "Tags" (listing linguistics, languages, email and messaging, human-computer interaction, and text and instant messaging).

# Spam Mail Data에 대한 이해 (Cont.)

## ■ Data Download :

- <https://www.kaggle.com/uciml/sms-spam-collection-dataset>

The screenshot shows the Kaggle dataset page for 'spam.csv'. The top section displays 'Data Sources' with a red box around 'spam.csv' (4 columns), 'About this file' (No description yet), and 'Columns' (v1, v2, A, A). Below this is a detailed preview of the 'spam.csv' file, showing its size (491.86 KB), 4 of 4 columns, and various views. The preview table has four columns: v1, v2, and two unnamed columns. The first row shows counts for 'ham' (87%) and 'spam' (13%), with 'spam' having 5169 unique values. The second row shows percentages for [null] (99%), 'bt not his girlfrnd....' (0%), 'MK17 92H. 450P...' (0%), and 'Other (42)' (1%). The third row shows percentages for 'Other (9)' (0%) and 'Other (9)' (0%). Below these are two rows of sample data: row 1 shows 'ham' with the message 'Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...', and row 2 shows 'ham' with the message 'Ok lar... Joking wif u oni...'. The preview table has 5 columns in total.

	v1	v2		
	ham 87%	5169 unique values	[null] 99%	[null] 100%
	spam 13%		bt not his girlfrnd.... 0%	MK17 92H. 450P... 0%
			Other (42) 1%	Other (9) 0%
1	ham	Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...		
2	ham	Ok lar... Joking wif u oni...		

# Spam Mail Data에 대한 이해 (Cont.)

```
1 import numpy as np  
2 import pandas as pd  
3 data = pd.read_csv('./spam.csv', encoding='latin1')
```

```
1 data[:5]
```

	v1	v2	Unnamed: 2	Unnamed: 3	Unnamed: 4
0	ham Go until jurong point, crazy.. Available only ...		NaN	NaN	NaN
1	ham Ok lar... Joking wif u oni...		NaN	NaN	NaN
2	spam Free entry in 2 a wkly comp to win FA Cup fina...		NaN	NaN	NaN
3	ham U dun say so early hor... U c already then say...		NaN	NaN	NaN
4	ham Nah I don't think he goes to usf, he lives aro...		NaN	NaN	NaN

## Spam Mail Data에 대한 이해 (Cont.)

- Unnamed라는 이름의 3개의 열은 텍스트 분류를 할 때 불필요한 열.
- **v1**열은 해당 Mail이 Spam 여부를 나타내는 Label
- **ham**은 정상 Mail 의미
- **spam**은 Spam Mail 의미
- **v2**열은 메일의 본문.
- Label과 Mail 내용이 담긴 **v1**열과 **v2**열만 필요하므로, Unnamed: 2, Unnamed: 3, Unnamed: 4 열 삭제
- **v1**열에 있는 ham과 spam Label을 각각 숫자 **0**과 **1**로 변환.

# Spam Mail Data에 대한 이해 (Cont.)

```
1 del data['Unnamed: 2']
2 del data['Unnamed: 3']
3 del data['Unnamed: 4']
4 data['v1'] = data['v1'].replace(['ham', 'spam'], [0, 1])
5 data[:5]
```

v1	v2
0 0	Go until jurong point, crazy.. Available only ...
1 0	Ok lar... Joking wif u oni...
2 1	Free entry in 2 a wkly comp to win FA Cup fina...
3 0	U dun say so early hor... U c already then say...
4 0	Nah I don't think he goes to usf, he lives aro...

# Spam Mail Data에 대한 이해 (Cont.)

## ■ Data 확인

```
1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5572 entries, 0 to 5571
Data columns (total 2 columns):
v1    5572 non-null int64
v2    5572 non-null object
dtypes: int64(1), object(1)
memory usage: 87.1+ KB
```

## ■ Null 값 여부 확인

```
1 data.isnull().values.any()
```

```
False
```

# Spam Mail Data에 대한 이해 (Cont.)

## ■ Data 중복 여부 확인

```
1 data['v2'].nunique(), data['v1'].nunique()
```

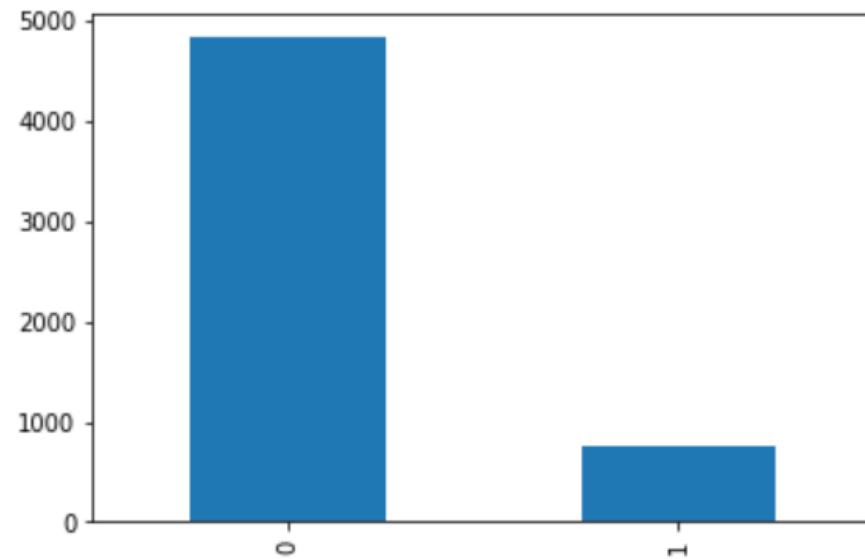
(5169, 2)

- 총 5,572개의 샘플 중 v2열에서 중복을 제거한 샘플의 개수가 5,169개이고, 403개의 중복 샘플이 존재한다는 의미.
- v1열은 **0** 또는 **1**의 값만을 가지므로 2가 출력되었다.

# Spam Mail Data에 대한 이해 (Cont.)

## ■ Spam Mail 유무를 의미하는 Label 값 분포의 시각화

```
1 import matplotlib.pyplot as plt  
2 %matplotlib inline  
3  
4 data['v1'].value_counts().plot(kind='bar');
```



# Spam Mail Data에 대한 이해 (Cont.)

- Label이 대부분 0에 편중되어 있는데, 이는 Spam Mail Data의 대부분의 Mail이 정상 Mail임을 의미

```
1 print(data.groupby('v1').size().reset_index(name='count'))
```

	v1	count
0	0	4825
1	1	747

- Label 0은 총 4,825개가 존재하고 1은 747개가 존재한다.
- x와 y를 분리한다. v2열을 x, v1열을 y로 저장한다.

```
1 X_data = data['v2']
2 y_data = data['v1']
3 print('메일 본문의 개수: %d' % len(X_data))
4 print('레이블의 개수: %d' % len(y_data))
```

메일 본문의 개수: 5572  
레이블의 개수: 5572

# Spam Mail Data에 대한 이해 (Cont.)

## ■ Tokenization and Integer Encoding

```
1 from keras.preprocessing.text import Tokenizer  
2 tokenizer = Tokenizer()  
3 tokenizer.fit_on_texts(X_data) #5572개의 행을 가진 X의 각 행에 토큰화를 수행  
4 sequences = tokenizer.texts_to_sequences(X_data) #단어를 숫자값, 인덱스로 변환하여 저장
```

Using TensorFlow backend.

```
1 print(sequences[:5])  
  
[[50, 469, 4410, 841, 751, 657, 64, 8, 1324, 89, 121, 349, 1325, 147, 2987, 1326, 67, 58, 4411, 144],  
[46, 336, 1495, 470, 6, 1929], [47, 486, 8, 19, 4, 796, 899, 2, 178, 1930, 1199, 658, 1931, 2320, 26  
7, 2321, 71, 1930, 2, 1932, 2, 337, 486, 554, 955, 73, 388, 179, 659, 389, 2988], [6, 245, 152, 23, 3  
79, 2989, 6, 140, 154, 57, 152], [1018, 1, 98, 107, 69, 487, 2, 956, 69, 1933, 218, 111, 471]]
```

■ 각 Mail에는 단어가 아니라 단어에 대한 Index가 부여됐다.

# Spam Mail Data에 대한 이해 (Cont.)

- **tokenizer.word\_index**는 **x\_data**에 존재하는 모든 단어와 부여된 Index를 리턴한다.

```
1 word_index = tokenizer.word_index
2 print(word_index)

{'i': 1, 'to': 2, 'you': 3, 'a': 4, 'the': 5, 'u': 6, 'and': 7, 'in': 8, 'is': 9, 'me': 10, 'my': 1
1, 'for': 12, 'your': 13, 'it': 14, 'of': 15, 'call': 16, 'have': 17, 'on': 18, '2': 19, 'that': 20,
'now': 21, 'are': 22, 'so': 23, 'but': 24, 'not': 25, 'or': 26, 'do': 27, 'can': 28, 'at': 29,
"i'm": 30, 'get': 31, 'be': 32, 'will': 33, 'if': 34, 'ur': 35, 'with': 36, 'just': 37, 'no': 38, 'w
e': 39, 'this': 40, 'gt': 41, '4': 42, 'It': 43, 'up': 44, 'when': 45, 'ok': 46, 'free': 47, 'from':
48, 'how': 49, 'go': 50, 'all': 51, 'out': 52, 'what': 53, 'know': 54, 'like': 55, 'good': 56, 'the
n': 57, 'got': 58, 'was': 59, 'come': 60, 'its': 61, 'am': 62, 'time': 63, 'only': 64, 'day': 65, 'I
ove': 66, 'there': 67, 'send': 68, 'he': 69, 'want': 70, 'text': 71, 'as': 72, 'txt': 73, 'one': 74,
'going': 75, 'by': 76, 'home': 77, "i'll": 78, 'need': 79, 'about': 80, 'r': 81, 'lor': 82, 'sorry':
83, 'stop': 84, 'still': 85, 'see': 86, 'back': 87, 'today': 88, 'n': 89, 'da': 90, 'our': 91, 'repl
y': 92, 'k': 93, 'dont': 94, 'she': 95, 'mobile': 96, 'take': 97, "don't": 98, 'tell': 99, 'hi': 10
0, 'new': 101, 'later': 102, 'her': 103, 'pls': 104, 'any': 105, 'please': 106, 'think': 107, 'bee
n': 108, 'they': 109, 'phone': 110, 'here': 111, 'week': 112, 'dear': 113, 'did': 114, 'some': 115,
'i': 116, 'I': 117, 'well': 118, 'has': 119, 'much': 120, 'great': 121, 'night': 122, 'oh': 123, 'cl
aim': 124, 'an': 125, 'hope': 126, 'hey': 127, 'msg': 128, 'who': 129, 'him': 130, 'where': 131,
'd': 132, 'more': 133, 'too': 134, 'happy': 135, 'had': 136, 'yes': 137, 'make': 138, 'way': 139,
'c': 140, 'www': 141, 'work': 142, 'give': 143, 'wat': 144, "it's": 145, 'number': 146, 'e': 147, 'm
essage': 148, 'should': 149, 'prize': 150, 'tomorrow': 151, 'say': 152, 'right': 153, 'already': 15
4, 'after': 155, 'ask': 156, 'cash': 157, 'doing': 158, 'said': 159, '3': 160, 'yeah': 161, 'reall
y': 162, 'amp': 163, 'whv': 164, 'im': 165, 'meet': 166, 'them': 167, 'life': 168, 'find': 169, 'ver
```

# Spam Mail Data에 대한 이해 (Cont.)

```
1 vocab_size = len(word_index)+1  
2 print('단어 집합의 크기 : %d' % (vocab_size))
```

단어 집합의 크기 : 8921

```
1 n_of_train = int(5572 * 0.8)  
2 n_of_test = int(5572 - n_of_train)  
3 print(n_of_train)  
4 print(n_of_test)
```

4457  
1115

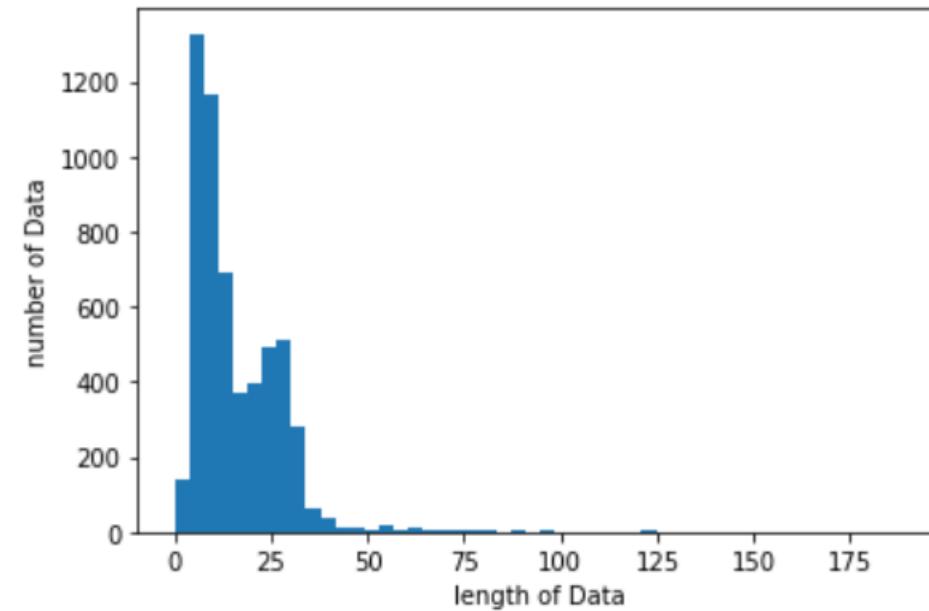
- 전체 메일의 개수는 5,572개
- 전체 데이터의 80%를 훈련용 Data로, 20%를 Test Data로 사용한다.
- 훈련 데이터는 4,457개, 테스트 데이터는 1,115개를 사용한다.

## Spam Mail Data에 대한 이해 (Cont.)

- 직관적으로 변수를 기억하기 위해 **x\_data**에 대해서 정수 Encoding 된 결과인 **sequences**를 **x\_data**로 변경한다.
- 전체 데이터에서 가장 길이가 긴 Mail과 전체 Mail Data의 길이 분포를 알아보도록 한다.

```
1 X_data=sequences
2 print('메일의 최대 길이 : %d' % max(len(l) for l in X_data))
3 print('메일의 평균 길이 : %f' % (sum(map(len, X_data))/len(X_data)))
4 plt.hist([len(s) for s in X_data], bins=50)
5 plt.xlabel('length of Data')
6 plt.ylabel('number of Data')
7 plt.show()
```

메일의 최대 길이 : 189  
메일의 평균 길이 : 15.794867



# RNN으로 Spam Mail 분류하기

```
1 from keras.layers import SimpleRNN, Embedding, Dense  
2 from keras.models import Sequential  
3 from keras.preprocessing.sequence import pad_sequences  
4 max_len = 189  
5 # 전체 DataSet의 길이는 189로 맞춘다. (메일의 최대 길이)  
6 data = pad_sequences(X_data, maxlen=max_len)  
7 print("data shape: ", data.shape)
```

data shape: (5572, 189)

- **maxlen**에는 가장 긴 메일의 길이였던 189이라는 숫자를 넣는다.
- 5,572개의 **x\_data**의 길이를 전부 189로 변환해준다.
- 189보다 길이가 짧은 Mail Sample은 전부 숫자 0이 Padding되어 189의 길이를 가지게 된다.

# RNN으로 Spam Mail 분류하기 (Cont.)

- 이제 **x\_train**과 **x\_test**를 분리한다.

```
1 X_test = data[n_of_train:] #X_data 데이터 중에서 뒤의 1115개의 데이터만 저장  
2 y_test = y_data[n_of_train:] #y_data 데이터 중에서 뒤의 1115개의 데이터만 저장  
3 X_train = data[:n_of_train] #X_data 데이터 중에서 앞의 4457개의 데이터만 저장  
4 y_train = y_data[:n_of_train] #y_data 데이터 중에서 앞의 4457개의 데이터만 저장
```

# RNN으로 Spam Mail 분류하기 (Cont.)

## ■ Model을 설계한다.

```
1 model = Sequential()
2 model.add(Embedding(vocab_size, 32)) # 임베딩 벡터의 차원은 32
3 model.add(SimpleRNN(32)) # RNN 셀의 hidden_size는 32
4 model.add(Dense(1, activation='sigmoid'))
5
6 model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
7 history = model.fit(X_train, y_train, epochs=4, batch_size=60, validation_split=0.2)
```

## ■ Embedding()의 2개의 parameter

- 단어 집합의 크기, Embedding Vector의 차원.
- 이진 분류(Binary Classification) 문제 이므로, 마지막 출력층에는 1개의 Neuron과 활성화 함수로 **Sigmoid** 함수를 사용한다.
- 손실 함수로는 **binary\_crossentropy**를 사용한다.

# RNN으로 Spam Mail 분류하기 (Cont.)

- 검증 데이터는 기계가 훈련 데이터에 과적합 되고 있지는 않은지 확인하기 위한 용도로 사용된다.
- 총 4번 학습한다.

Train on 3565 samples, validate on 892 samples

Epoch 1/4

3565/3565 [=====] - 7s 2ms/step - loss: 0.3009 - acc: 0.8912 - val\_loss: 0.1083 - val\_acc: 0.9731

Epoch 2/4

3565/3565 [=====] - 5s 1ms/step - loss: 0.0846 - acc: 0.9764 - val\_loss: 0.0553 - val\_acc: 0.9843

Epoch 3/4

3565/3565 [=====] - 5s 1ms/step - loss: 0.0427 - acc: 0.9888 - val\_loss: 0.0778 - val\_acc: 0.9731

Epoch 4/4

3565/3565 [=====] - 5s 1ms/step - loss: 0.0744 - acc: 0.9734 - val\_loss: 0.0661 - val\_acc: 0.9821

# RNN으로 Spam Mail 분류하기 (Cont.)

- Test Data에 대해서 정확도를 확인해 본다.

```
1 print("테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
```

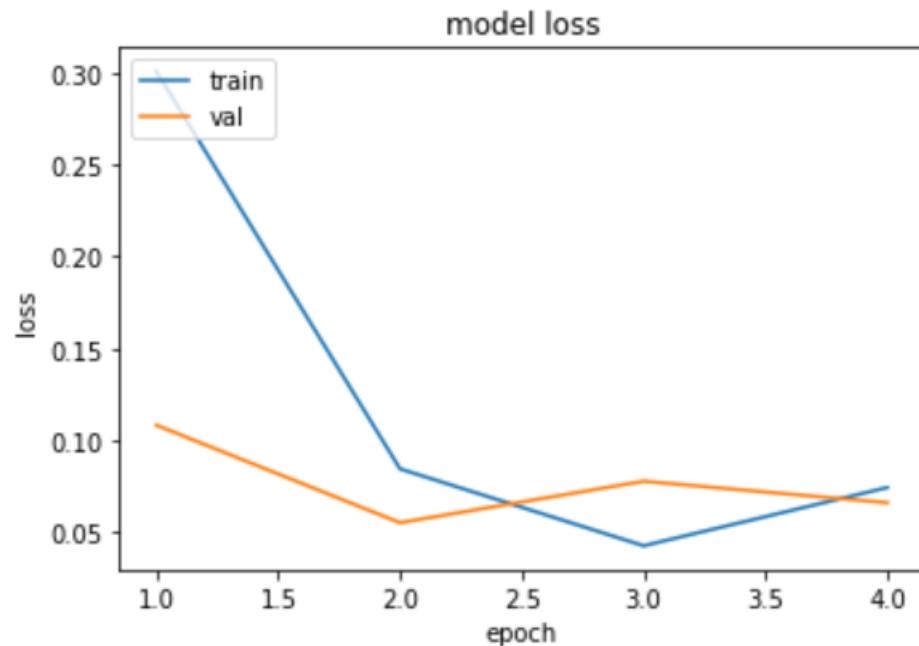
```
1115/1115 [=====] - 1s 524us/step
```

```
테스트 정확도: 0.9821
```

# RNN으로 Spam Mail 분류하기 (Cont.)

- Graph로 시각화해 본다.

```
1 epochs = range(1, len(history.history['acc']) + 1)
2 plt.plot(epochs, history.history['loss'])
3 plt.plot(epochs, history.history['val_loss'])
4 plt.title('model loss')
5 plt.ylabel('loss')
6 plt.xlabel('epoch')
7 plt.legend(['train', 'val'], loc='upper left')
8 plt.show()
```



## RNN으로 Spam Mail 분류하기 (Cont.)

- 만일 이번 실습처럼 Data의 양이 적으면 과적합이 발생할 수 있으므로 검증 데이터에 대한 오차가 증가하기 시작하는 시점의 바로 직전인 Epoch를 3~4 정도가 적당한다.
- 이 Data는 Epoch 5를 넘어가기 시작하면 검증 데이터의 오차가 증가하는 경향이 있다.

---

### **3. Reuters News Classification**



# Reuters News Classification

## ■ Data

- Keras에서 제공하는 Reuters News
- 총 11,258개의 News 기사, 46개의 News Category로 분류되는 뉴스 기사 Data

## ■ Model

- LSTM

## ■ 목적

- Text Classification

# Reuters New Data에 대한 이해

## ■ Keras Dataset으로부터 Reuters 기사 Data Downloads

```
1 from keras.datasets import reuters  
2 (X_train, y_train), (X_test, y_test) = reuters.load_data(num_words=None, test_split=0.2)
```

Using TensorFlow backend.

Downloading data from <https://s3.amazonaws.com/text-datasets/reuters.npz>  
2113536/2110848 [=====] - 2s 1us/step

## ■ num\_words

- 데이터에서 등장 빈도 순위
- 몇 번째에 해당하는 단어까지를 갖고 올 것인지 조절
- 예) 100이란 값을 넣으면, 등장 빈도 순위가 1~100에 해당하는 단어만 갖고 오게 된다.
- 예) None : 모든 단어를 사용

# Reuters New Data에 대한 이해 (Cont.)

## ■ test\_split

- 전체 뉴스 기사 데이터 중 Test용 뉴스 기사로 몇 Percent를 사용할 것인지 결정
- 예) 0.2 → 전체 뉴스 기사 중 20%를 Test용 뉴스 기사로 사용한다는 의미

```
1 print('훈련용 뉴스 기사: {}'.format(len(X_train)))
2 print('테스트용 뉴스 기사: {}'.format(len(X_test)))
3 num_classes = max(y_train) + 1
4 print('카테고리: {}'.format(num_classes))
```

훈련용 뉴스 기사: 8982

테스트용 뉴스 기사: 2246

카테고리: 46

## ■ y\_train

- 0부터 시작하는 숫자들
- 가장 큰 수에 +1을 하여 출력하면 Category가 총 몇 개인지를 알 수 있다.

# Reuters New Data에 대한 이해 (Cont.)

- 훈련용 뉴스 기사 데이터의 구성을 확인하기 위해 첫 번째 News 기사 출력.

```
1 print(X_train[0]) # 첫번째 훈련용 뉴스 기사  
2 print(y_train[0]) # 첫번째 훈련용 뉴스 기사의 레이블
```

```
[1, 27595, 28842, 8, 43, 10, 447, 5, 25, 207, 270, 5, 3095, 111, 16, 369, 186, 90, 67, 7, 89, 5, 19,  
102, 6, 19, 124, 15, 90, 67, 84, 22, 482, 26, 7, 48, 4, 49, 8, 864, 39, 209, 154, 6, 151, 6, 83, 11,  
15, 22, 155, 11, 15, 7, 48, 9, 4579, 1005, 504, 6, 258, 6, 272, 11, 15, 22, 134, 44, 11, 15, 16, 8, 1  
97, 1245, 90, 67, 52, 29, 209, 30, 32, 132, 6, 109, 15, 17, 12]  
3
```

# Reuters New Data에 대한 이해 (Cont.)

## ■ 훈련용 뉴스 기사 데이터 **x\_train**

- 정수의 나열이 저장되어 있다.
- Data는 Tokenizing과 Integer Encoding이 끝난 Data이다.
- 단어들이 몇 번 등장하는가의 빈도순에 따라서 Index 부여
- 1이라는 숫자는 이 단어가 이 데이터에서 등장 빈도가 첫 번째로 많다는 뜻.
- 따라서 뒤로 갈 수록 빈도가 적은 단어이다.
- 만약 **num\_words**에 1,000을 넣었다면 빈도수 순위가 1,000 이하의 단어만 가져온다는 의미이므로 Data에서 1,000을 넘는 정수는 나오지 않게 된다.

## ■ 뉴스 기사들의 Label **y\_train**

- 3이라는 값은 첫 번째 훈련용 News 기사가 46개의 Category 중 3에 해당

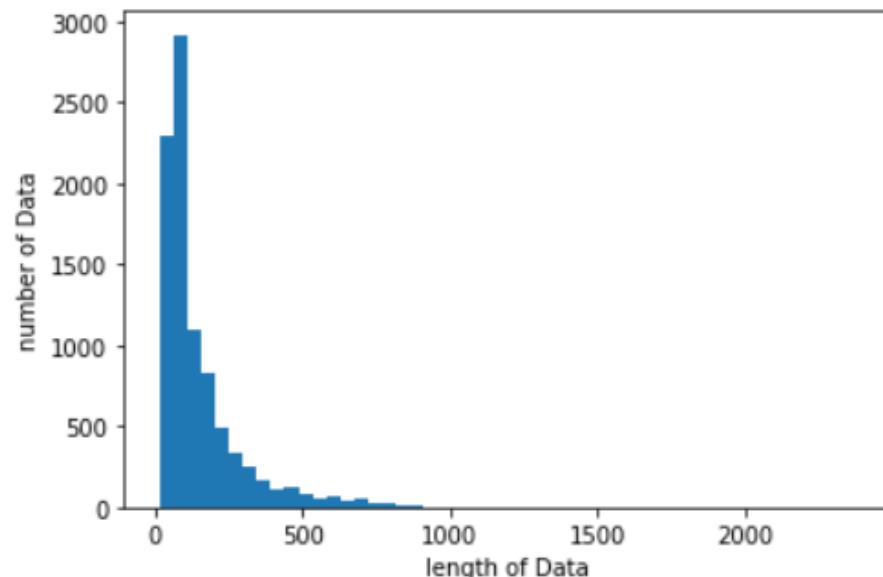
# Reuters New Data에 대한 이해 (Cont.)

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3
4 print('뉴스 기사의 최대 길이 :',max(len(l) for l in X_train))
5 print('뉴스 기사의 평균 길이 :',sum(map(len, X_train))/len(X_train))
6
7 plt.hist([len(s) for s in X_train], bins=50)
8 plt.xlabel('length of Data')
9 plt.ylabel('number of Data')
10 plt.show()
```

뉴스 기사의 최대 길이 : 2376

뉴스 기사의 평균 길이 : 145.5398574927633

대부분의 News는 100 ~ 200  
사이의 길이를 가짐

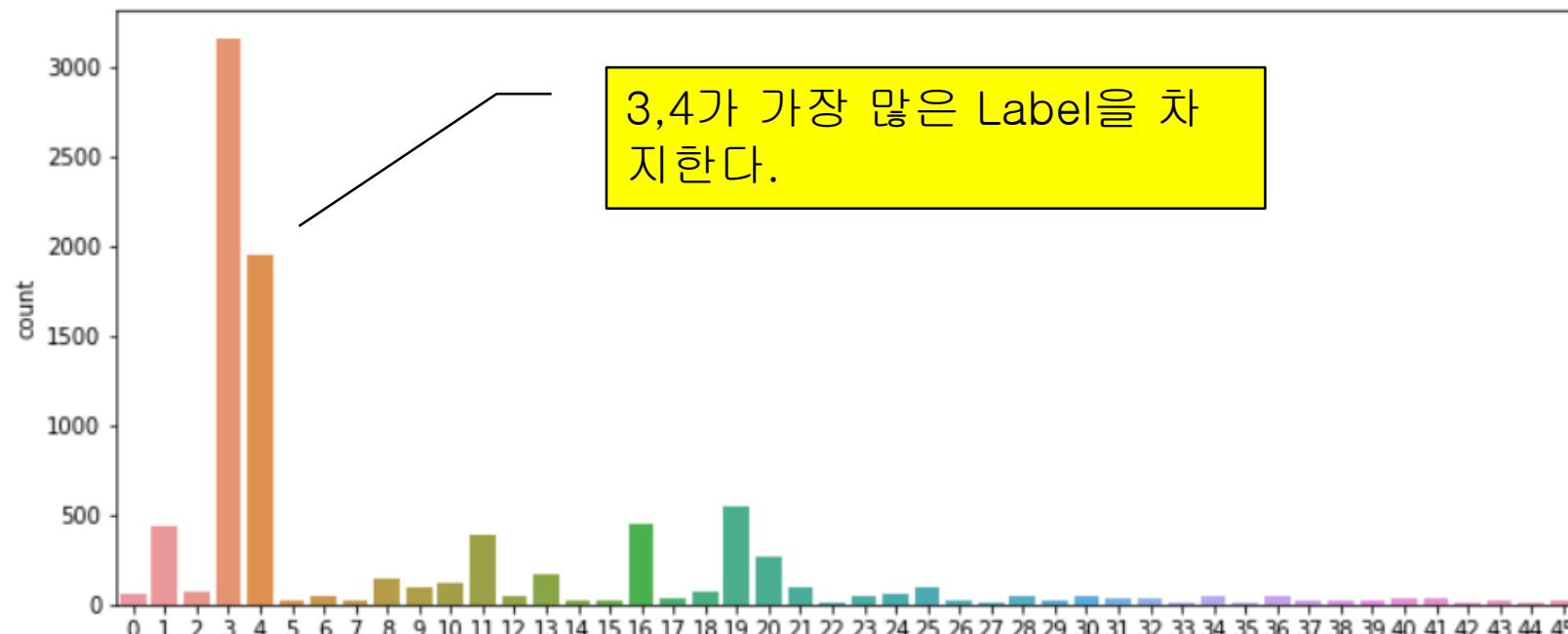


# Reuters New Data에 대한 이해 (Cont.)

- 각 뉴스가 어떤 종류의 뉴스에 속하는지 기재되어있는 Label 값의 분포를 보도록 한다.

```
1 import seaborn as sns  
2 fig, axe=plt.subplots(ncols=1)  
3 fig.set_size_inches(12,5)  
4 sns.countplot(y_train)
```

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f7eb93604a8>



# Reuters New Data에 대한 이해 (Cont.)

## ■ 각 Label에 대한 정확한 개수.

```
1 import numpy as np
2 unique_elements, counts_elements = np.unique(y_train, return_counts=True)
3 print("각 레이블에 대한 빈도수:")
4 print(np.asarray((unique_elements, counts_elements)))
5 # label_cnt=dict(zip(unique_elements, counts_elements))
6 # 아래의 출력 결과가 보기 불편하여 병렬로 보고싶다면 위의 label_cnt를 출력
```

각 레이블에 대한 빈도수:

```
[[ 0   1   2   3   4   5   6   7   8   9   10  11  12  13
  14  15  16  17  18  19  20  21  22  23  24  25  26  27
  28  29  30  31  32  33  34  35  36  37  38  39  40  41
  42  43  44  45]
 [ 55  432   74  3159  1949   17   48   16   139   101   124   390
   49   172
  26   20  444   39   66  549  269  100   15   41   62   92   24
   15
  48   19   45   39   32   11   50   10   49   19   19   24   36
   30
  13   21   12   18]]
```

# Reuters New Data에 대한 이해 (Cont.)

- x\_train에 들어있는 숫자들이 각자 어떤 단어들을 나타내고 있는지 확인해본다.

```
1 word_index = reuters.get_word_index()  
2 print(word_index)
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/reuters_word_index.json  
557056/550378 [=====] - 1s 2us/step  
{'mdb1': 10996, 'fawc': 16260, 'degussa': 12089, 'woods': 8803, 'hanging': 13796, 'localized': 2067  
2, 'sation': 20673, 'chanthaburi': 20675, 'refunding': 10997, 'hermann': 8804, 'passengers': 20676,  
'stipulate': 20677, 'heublein': 8352, 'screaming': 20713, 'tcby': 16261, 'four': 185, 'grains': 164  
2, 'broiler': 20680, 'wooden': 12090, 'wednesday': 1220, 'highveld': 13797, 'duffour': 7593, '0053':  
20681, 'elections': 3914, '270': 2563, '271': 3551, '272': 5113, '273': 3552, '274': 3400, 'rudman':  
7975, '276': 3401, '277': 3478, '278': 3632, '279': 4309, 'dormancy': 9381, 'errors': 7247, 'deferred':  
3086, 'sptnd': 20683, 'cooking': 8805, 'stratabit': 20684, 'designing': 16262, 'metalurgicos': 2  
0685, 'databank': 13798, '300er': 20686, 'shocks': 20687, 'nawg': 7972, 'tnfa': 20688, 'perforations':  
20689, 'affiliates': 2891, '27p': 20690, 'ching': 16263, 'china': 595, 'wagyu': 16264, 'affiliated':  
3189, 'chino': 16265, 'chinh': 16266, 'slickline': 20692, 'doldrums': 13799, 'kids': 12092, 'climbed':  
3028, 'controversy': 6693, 'kidd': 20693, 'spotty': 12093, 'rebel': 12639, 'millimetres': 93  
82, 'golden': 4007, 'projection': 5689, 'stern': 12094, "hudson's": 7903, 'dna': 10066, 'dnc': 2069  
5, 'hodler': 20696, 'lme': 2394, 'insolvency': 20697, 'music': 13800, 'therefore': 1984, 'dns': 1099  
8, 'distortions': 6959, 'thassos': 13801, 'populations': 20698, 'meteorologist': 8806, 'loss': 43,  
'exco': 9383, 'adventist': 20813, 'murchison': 16267, 'locked': 10999, 'kampala': 13802, 'arndt': 20  
699, 'nakasone': 1267, 'steinweg': 20700, "india's": 3633, 'wang': 3029, 'wane': 10067, 'unjust': 13  
803, 'titanium': 13804, 'want': 850, 'pinto': 20701, "institutes)": 16268, 'absolute': 7973, 'trave  
l': 4677, 'cutback': 6422, 'nazmi': 16269, 'modest': 1858, 'shonwell': 16270, 'sedi': 20702, 'adone
```

## Reuters New Data에 대한 이해 (Cont.)

- `x_train`에 들어있는 숫자들이 각자 어떤 단어들을 나타내고 있는지 다른 방법으로 확인해본다.
- 즉, Index로부터 단어를 알 수 있도록 출력하는 방법이다.

```
1 index_to_word={}
2 for key, value in word_index.items():
3     index_to_word[value] = key
```

```
1 print(index_to_word[28842])
```

nondiscriminatory

- 28842번째 단어는 `nondiscriminatory`임을 알 수 있다.

# Reuters New Data에 대한 이해 (Cont.)

- 이 데이터에서 가장 많이 등장하는 단어는 ?

```
1 print(index_to_word[1])
```

the

- **index\_to\_word**를 이용해서 첫 번째 훈련용 뉴스 기사인 **x\_train[0]**가 어떤 단어들로 구성되어있는지를 복원하자.

```
1 print(' '.join([index_to_word[X] for X in x_train[0]]))
```

the wattie nondiscriminatory mln loss for plc said at only ended said commonwealth could 1 traders no w april 0 a after said from 1985 and from foreign 000 april 0 prices its account year a but in this m ln home an states earlier and rise and revs vs 000 its 16 vs 000 a but 3 psbr oils several and shareh olders and dividend vs 000 its all 4 vs 000 1 mln agreed largely april 0 are 2 states will billion to tal and against 000 pct dtrs

# LSTM으로 로이터 뉴스 분류하기

## 1. Libraries Loading

```
1 from keras.datasets import reuters  
2 from keras.models import Sequential  
3 from keras.layers import Dense, LSTM, Embedding  
4 from keras.preprocessing import sequence  
5 from keras.utils import np_utils
```

## 2. 등장 빈도가 1000번째 까지만 사용

```
1 (X_train, y_train), (X_test, y_test) = reuters.load_data(num_words=1000, test_split=0.2)
```

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

- 각 기사는 단어의 수가 서로 다르다.
- 3. Model의 입력으로 사용하고자 모든 뉴스 기사의 길이를 동일하게 맞춘다.
- **pad\_sequences()**를 사용하여 **maxlen**의 값으로 100을 설정함으로, 모든 News 기사의 길이. 즉, 단어 수를 100으로 일치시킨다.
- 만일 단어의 개수가 100개보다 많으면 100개만 선택하고 나머지는 제거하며, 100개보다 부족한 경우에는 부족한 부분이 0으로 패딩된다.

```
1 max_len=100
2 X_train = sequence.pad_sequences(X_train, maxlen=max_len) # 훈련용 뉴스 기사 패딩
3 X_test = sequence.pad_sequences(X_test, maxlen=max_len) # 테스트용 뉴스 기사 패딩
```

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

4. 훈련용, Test용 뉴스 기사 데이터의 Label에 One-hot Encoding 수행하기.

```
1 y_train = np_utils.to_categorical(y_train) # 훈련용 뉴스 기사 레이블의 원-핫 인코딩  
2 y_test = np_utils.to_categorical(y_test) # 테스트용 뉴스 기사 레이블의 원-핫 인코딩
```

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

## 5. LSTM 모델 생성.

- Embedding()을 사용하여 Embedding Layer 생성
- 최소 두 개의 인자
- 첫 번째 인자는 단어 집합의 크기
- 두 번째 인자는 Embedding Vector의 차원.

```
1 model = Sequential()
2 model.add(Embedding(1000, 120))
3 model.add(LSTM(120))
4 model.add(Dense(46, activation='softmax'))
```

- 위 코드에서 120의 차원을 가지는 Embedding Vector를 1,000개 생성
- 120차원의 Embedding Vector들을 LSTM에 할당한다.
  - LSTM의 인자는 Memory Cell의 은닉 상태의 크기(**hidden\_size**).
- 출력층에서는 46개의 Neuron 사용
  - 활성화 함수로 **Softmax** 함수 사용

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

## 6. Model Compile

- Multi-Class Classification 문제이므로 손실 함수로는 **categorical\_crossentropy**를 사용.
- **categorical\_crossentropy**는 모델의 예측값과 실제값에 대해서 두 확률 분포 사이의 거리를 최소화하도록 훈련한다.

```
1 model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

## 7. Model 학습하기

- **validation\_data**로 **x\_test**와 **y\_test**를 사용한다.
- **val\_loss**가 줄어들다가 증가하는 상황이 오면 과적합(overfitting)으로 판단하기 위함이다.

```
1 history = model.fit(X_train, y_train, batch_size=100, epochs=20, validation_data=(X_test,y_test))
```

```
Epoch 16/20
8982/8982 [=====] - 20s 2ms/step - loss: 0.8586 - acc: 0.7792 - val_loss: 1.
1760 - val_acc: 0.7115
Epoch 17/20
8982/8982 [=====] - 21s 2ms/step - loss: 0.8153 - acc: 0.7926 - val_loss: 1.
1786 - val_acc: 0.7066
Epoch 18/20
8982/8982 [=====] - 21s 2ms/step - loss: 0.7841 - acc: 0.7995 - val_loss: 1.
1705 - val_acc: 0.7084
Epoch 19/20
8982/8982 [=====] - 21s 2ms/step - loss: 0.7447 - acc: 0.8111 - val_loss: 1.
1421 - val_acc: 0.7177
Epoch 20/20
8982/8982 [=====] - 21s 2ms/step - loss: 0.7055 - acc: 0.8202 - val_loss: 1.
2115 - val_acc: 0.7070
```

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

## 8. Test Data에 대한 정확도 체크

```
1 print("테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
```

```
2246/2246 [=====] - 2s 805us/step
```

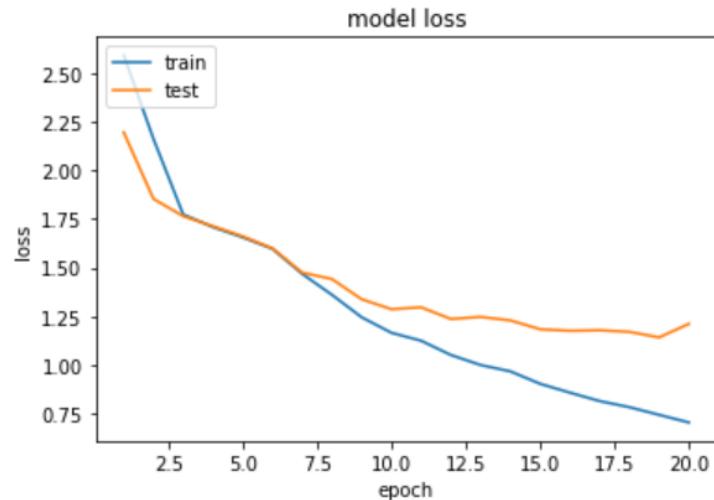
테스트 정확도: 0.7070

- 70%의 정확도를 얻었다.
- `model.fit()`에서 **validation\_data**는 실제 훈련에는 반영되지 않고 과적합을 판단하기 위한 용도로만 사용된다.
- **validation\_data**는 기계의 학습에 반영하지 않고 오직 정확도의 결과로만 보여준다.
- 그러므로 **validation\_data**에 **x\_test, y\_test**를 사용하더라도 기계는 아직 이 데이터로 학습한 적이 없는 상태이기 때문에, Model이 학습하지 않은 데이터로 정확도를 확인하기 위한 용도인 테스트 정확도를 위한 용도로도 사용하였다.

# LSTM으로 로이터 뉴스 분류하기 (Cont.)

## 8. 그래프로 시각화

```
1 epochs = range(1, len(history.history['acc']) + 1)
2 plt.plot(epochs, history.history['loss'])
3 plt.plot(epochs, history.history['val_loss'])
4 plt.title('model loss')
5 plt.ylabel('loss')
6 plt.xlabel('epoch')
7 plt.legend(['train', 'test'], loc='upper left')
8 plt.show()
```



- 검증 데이터에서의 오차가 줄어들고 있는 것을 확인할 수 있다.
- 하지만 검증 데이터와 훈련 데이터의 오차의 차이가 벌어지고 있는 것은 과적합의 신호일 수 있으므로 주의해야 한다.

---

## 4. IMDB Movie Review Sentiment Analysis



# IMDB Movie Review Sentiment Analysis

- Machine Learning에서 텍스트 분류 중 특히, 감성 분류를 연습하기 위해 자주 사용하는 데이터.
- Standford Univ.에서 2011년 논문에서 소개
- 당시 논문에서는 이 데이터를 훈련 데이터와 테스트 데이터를 50:50대 비율로 분할하여 88.89%의 정확도를 얻었다고 소개하고 있다.
- 영화 사이트 IMDB의 리뷰 데이터.
- Movie Review에 대한 Text와 Review가 긍정인 경우 **1**, 부정인 경우 **0**으로 표시한 Label로 구성된 데이터.
- 논문 링크 : [http://ai.stanford.edu/~amaas/papers/wvSent\\_acl2011.pdf](http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf)
- Keras에서는 해당 IMDB Movie Review Data를 **imdb.load\_data()** 함수를 통해 Download.

# IMDB Review Data에 대한 이해

- Movie Review Data Load
- Keras에서 제공하는 IMDB Review Data는 이미 Train Data와 Test Data를 50:50 비율로 구분해서 제공한다.
- **num\_words**는 Data에서 등장 빈도 순위를 의미
  - 몇 번째에 해당하는 단어까지를 갖고 올 것인지 조절하는 것
  - 예) 10,000이란 값을 넣으면, 등장 빈도 순위가 1~10,000에 해당하는 단어만 Loading 된다.
  - 단어의 종류는 10,000개가 되므로 단어 집합의 크기는 10,000이 된다.

```
1 from keras.datasets import imdb  
2 (X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)
```

Using TensorFlow backend.

Downloading data from <https://s3.amazonaws.com/text-datasets/imdb.npz>  
17465344/17464789 [=====] - 7s 0us/step

# IMDB Review Data에 대한 이해 (Cont.)

```
1 print('훈련용 리뷰 개수: {}'.format(len(X_train)))
2 print('테스트용 리뷰 개수: {}'.format(len(X_test)))
3 num_classes = max(y_train) + 1
4 print('카테고리: {}'.format(num_classes))
```

훈련용 리뷰 개수: 25000

테스트용 리뷰 개수: 25000

카테고리: 2

- **y\_train**은 0부터 시작해서 레이블을 부여하므로, **y\_train**에 들어 있는 가장 큰 수에 +1을 하여 출력함으로 Category가 총 몇 개인지를 알 수 있다.

# IMDB Review Data에 대한 이해 (Cont.)

```
1 print(X_train[0])
2 print(y_train[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 8
38, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111,
17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 1
2, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316,
8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 4
8, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256, 4,
2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 2
97, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535,
18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 3
2, 15, 16, 5345, 19, 178, 32]
1
```

- 첫 번째 훈련용 리뷰(**x\_train[0]**, **y\_train[0]**)에서 Review 본문에 해당하는 **x\_train[0]**에는 숫자들이 들어있다.
- 이 Data는 Tokenizing과 Integer Encoding이라는 Text Preprocessing가 끝난 상태이다.

## IMDB Review Data에 대한 이해 (Cont.)

- IMDB Review Data는 전체 Data에서 각 단어들이 몇 번 등장하는 지의 빈도에 따라서 Index를 부여했다.
- 1이라는 숫자는 이 단어가 이 Data에서 등장 빈도가 1등이라는 뜻이다.
- 따라서 973라는 숫자는 이 단어가 데이터에서 973번째로 빈도수가 높은 단어라는 뜻이 된다.
- 첫 번째 훈련용 Review에서 Label에 해당하는 `y_train[0]`은 1이라는 값이 들어다.
- 이 숫자는 첫 번째 훈련 Data가 2개의 Category 중 1에 해당하는 Category임을 의미한다.
- 이 예제의 경우 감성 정보로서 0 또는 1의 값을 가지는데, 이 경우에는 긍정을 의미하는 1의 값을 가진 것을 의미하고 있다.

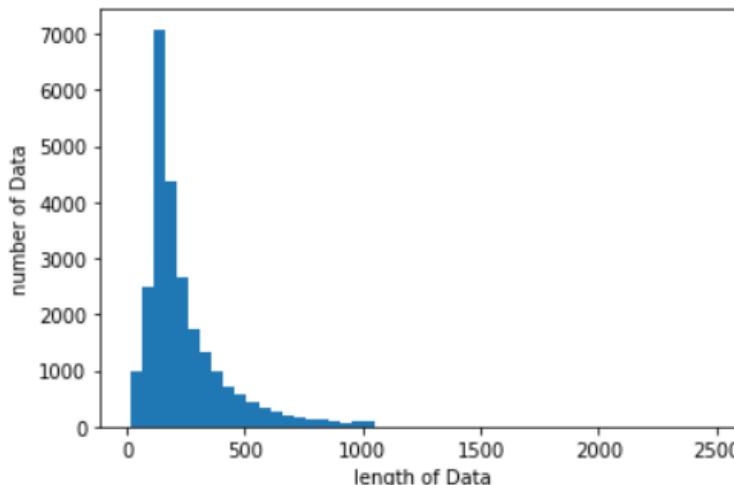
# IMDB Review Data에 대한 이해 (Cont.)

- 25,000개의 훈련용 Review의 각 길이는 전부 다르다.
- Review 각 길이가 대체적으로 어떤 크기를 가지는지 Graph를 통해 확인해 보도록 하자.

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3
4 print('리뷰의 최대 길이 :',max(len(l) for l in X_train))
5 print('리뷰의 평균 길이 :',sum(map(len, X_train))/len(X_train))
6
7 plt.hist([len(s) for s in X_train], bins=50)
8 plt.xlabel('length of Data')
9 plt.ylabel('number of Data')
10 plt.show()
```

리뷰의 최대 길이 : 2494

리뷰의 평균 길이 : 238.71364



## IMDB Review Data에 대한 이해 (Cont.)

- 대체적으로 500이하의 길이이다.
- 특히 100~300길이를 가진 데이터가 많은 것을 확인할 수 있다.
- 반면, 가장 긴 길이를 가진 데이터는 길이가 1,000이 넘는 것도 확인할 수 있다.
- 이제 Label의 분포를 확인해보자.

```
1 import numpy as np
2 unique_elements, counts_elements = np.unique(y_train, return_counts=True)
3 print("각 레이블에 대한 빈도수:")
4 print(np.asarray((unique_elements, counts_elements)))
```

각 레이블에 대한 빈도수:

```
[[    0     1]
 [12500 12500]]
```

## IMDB Review Data에 대한 이해 (Cont.)

- 25,000개의 Review가 있다.
- 두 Label 0과 1은 각각 12,500개로 균등한 분포를 가지고 있다.
- **x\_train**에 들어있는 숫자들이 각각 어떤 단어들을 나타내고 있는지 확인해보자.

```
1 word_index = imdb.get_word_index()  
2 index_to_word={}  
3 for key, value in word_index.items():  
4     index_to_word[value] = key
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/imdb\_word\_index.json  
1646592/1641221 [=====] - 1s 1us/step
```

## IMDB Review Data에 대한 이해 (Cont.)

- **index\_to\_word**에 Index를 집어 넣으면 전처리 전에 어떤 단어였는지 확인할 수 있다.

```
1 print(index_to_word[1])
```

the

```
1 print(index_to_word[3941])
```

journalist

- 위의 결과에서 이 Data에서는 빈도가 가장 높은 단어는 **the**이고, 빈도가 3941번째로 높은 단어는 **journalist**임을 알 수 있다.

# IMDB Review Data에 대한 이해 (Cont.)

- 첫 번째 훈련용 Review의 **x\_train[0]**이 Index로 바뀌기 전에 어떤 단어들이었는지 확인해볼 수 있다.

```
1 print(' '.join([index_to_word[X] for X in X_train[0]]))
```

the as you with out themselves powerful lets loves their becomes reaching had journalist of lot from anyone to have after out atmosphere never more room and it so heart shows to years of every never going and help moments or of every chest visual movie except her was several of enough more with is now current film as you of mine potentially unfortunately of you than him that with out themselves her get for was camp of you movie sometimes movie that with scary but and to story wonderful that in seeing in character to of 70s musicians with heart had shadows they of here that with her serious to have does when from why what have critics they is you that isn't one will very to as itself with other and in of seen over landed for anyone of and br show's to whether from than out themselves history he name half some br of and odd was two most of mean for 1 any an boat she he should is thought frog but of script you not while history he heart to real at barrel but when from one bit then have two of script their with her nobody most that with wasn't to with armed acting watch an for with heartfelt film want an

# LSTM으로 IMDB Review 감성 분류하기

## 1. Packages Loading

```
1 from keras.datasets import imdb  
2 from keras.models import Sequential  
3 from keras.layers import Dense, LSTM, Embedding  
4 from keras.preprocessing import sequence
```

## 2. Train Data는 빈도 순위가 5,000 이상인 단어들만 사용한다.

```
1 (X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=5000)
```

# LSTM으로 IMDB Review 감성 분류하기 (Cont.)

3. 각 Review는 문장의 길이가 다르기 때문에, Model이 처리할 수 있도록 길이를 동일하게 해주어야 한다.

- **pad\_sequences()** 사용
- 길이는 **maxlen**에 넣는 값으로 정해진다.
- Train Data가 정한 길이를 초과하면 초과분을 삭제하고, 부족하면 0으로 채운다.
- 여기서는 **maxlen**의 값을 500으로 지정한다.

```
1 max_len=500
2 X_train = sequence.pad_sequences(X_train, maxlen=max_len)
3 X_test = sequence.pad_sequences(X_test, maxlen=max_len)
```

# LSTM으로 IMDB Review 감성 분류하기 (Cont.)

## 4. Model 생성

- **Embedding()**

- 첫 번째 인자는 단어 집합의 크기
- 두 번째 인자는 Embedding Vector 크기
- 입력 Data에서 모든 단어는 120 차원을 가진 Embedding Vector로 표현됩니다.
- 출력층은 활성화 함수로 Sigmoid 함수를 갖는 Neuron 하나를 사용

```
1 model = Sequential()
2 model.add(Embedding(5000, 120))
3 model.add(LSTM(120))
4 model.add(Dense(1, activation='sigmoid'))
```

# LSTM으로 IMDB Review 감성 분류하기 (Cont.)

## 5. Compile

- 손실 함수는 **binary\_crossentropy**를 사용
- 최적화 함수는 **adam**을 사용.
- Epoch마다 정확도를 구하기 위해 **accuracy**를 추가.
- Epoch는 총 5번을 수행

```
1 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
2 model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5, batch_size=64)
3 scores = model.evaluate(X_test, y_test, verbose=0) # 테스트 데이터에 대해서 정확도 평가
4 print("정확도: %.2f%%" % (scores[1]*100))
```

# LSTM으로 IMDB Review 감성 분류하기 (Cont.)

- **validation\_data**에 Test Data인 **x\_test**와 **y\_test**를 사용할 경우에는 마지막 Epoch포크에서의 **val\_acc**가 Test Data의 정확도이다.
- Test Data에 대해서 87%의 정확도가 나왔다.

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/5
25000/25000 [=====] - 383s 15ms/step - loss: 0.4615 - acc: 0.7766 - val_loss: 0.3523 - val_acc: 0.8515
Epoch 2/5
25000/25000 [=====] - 386s 15ms/step - loss: 0.3682 - acc: 0.8364 - val_loss: 0.4580 - val_acc: 0.7823
Epoch 3/5
25000/25000 [=====] - 385s 15ms/step - loss: 0.3216 - acc: 0.8647 - val_loss: 0.3593 - val_acc: 0.8434
Epoch 4/5
25000/25000 [=====] - 389s 16ms/step - loss: 0.2491 - acc: 0.9031 - val_loss: 0.3435 - val_acc: 0.8621
Epoch 5/5
25000/25000 [=====] - 387s 15ms/step - loss: 0.2023 - acc: 0.9228 - val_loss: 0.3749 - val_acc: 0.8624
정확도: 86.24%
```

# 5. Naïve Bayes Classifier



# Naïve Bayes Classifier

## ■ 소개

- Text Classification를 위해 전통적으로 사용되는 분류기로 Naïve Bayes 분류기가 있다.
- Naïve Bayes 분류기는 ANN Algorithm에는 속하지 않지만, Machine Learning의 주요 Algorithm으로 분류에 있어 준수한 성능을 보여주는 것으로 알려져 있다.

# Bayes' theorem을 이용한 분류 Mechanism

■ Bayes' theorem은 조건부 확률을 계산하는 방법 중 하나이다.

■  $P(A)$

- A가 일어날 확률

■  $P(B)$

- B가 일어날 확률

■  $P(B|A)$

- A가 일어나고나서 B가 일어날 확률

■  $P(A|B)$

- B가 일어나고나서 A가 일어날 확률

## Bayes' theorem을 이용한 분류 Mechanism (Cont.)

- $P(B|A)$  를 쉽게 구할 수 있는 상황이라면, 아래와 같은 식을 통해  $P(A|B)$  를 구할 수 있다.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- Naïve Bayes 분류기는 이러한 Bayes' theorem를 이용하여 Text 분류를 수행한다.

## Bayes' theorem을 이용한 분류 Mechanism (Cont.)

- 예) Naïve Bayes 분류기를 통해서 Spam Mail Filter를 만들어본다고 하면, 입력 Text(Mail의 본문)이 주어졌을 때, 입력 Text가 정상 Mail인지 Spam Mail인지 구분하기 위한 확률을 표현할 수 있다.

$P(\text{정상 Mail} | \text{입력 Text}) = \text{입력 Text가 있을 때 정상 Mail일 확률}$

$P(\text{Spam Mail} | \text{입력 Text}) = \text{입력 Text가 있을 때 Spam Mail일 확률}$

- 이를 Bayes' theorem에 따라서 식을 표현하면 아래와 같다.

$P(\text{정상 Mail} | \text{입력 Text}) = (P(\text{입력 Text} | \text{정상 Mail}) \times P(\text{정상 Mail})) / P(\text{입력 Text})$

$P(\text{Spam Mail} | \text{입력 Text}) = (P(\text{입력 Text} | \text{Spam Mail}) \times P(\text{Spam Mail})) / P(\text{입력 Text})$

- 식을 보다 간소화할 수 있다.

$P(\text{정상 Mail} | \text{입력 Text}) = P(\text{입력 Text} | \text{정상 Mail}) \times P(\text{정상 Mail})$

$P(\text{Spam Mail} | \text{입력 Text}) = P(\text{입력 Text} | \text{Spam Mail}) \times P(\text{Spam Mail})$

## Bayes' theorem을 이용한 분류 Mechanism (Cont.)

- Mail의 본문에 있는 모든 단어를 Token化 시켜서 이 단어들을 Naïve Bayes의 분류기의 입력으로 사용한다.
- Naïve Bayes 분류기에서 Token화 이전의 단어의 순서는 중요하지 않다.
- BoW와 같이 단어의 순서를 무시하고 오직 빈도수만을 고려한다.

# Spam Detection

## ■ Train Data

-	메일로부터 토큰화 및 정제 된 단어들	분류
1	me free lottery	스팸 메일
2	free get free you	스팸 메일
3	you free scholarship	정상 메일
4	free to contact me	정상 메일
5	you won award	정상 메일
6	you ticket lottery	스팸 메일

## Spam Detection (Cont.)

- 앞의 Train Data에서 *you free lottery*라는 입력 Text에 대해서 정상 Mail일 확률과 Spam Mail일 확률을 계산해본다.

$P(\text{정상 Mail} | \text{입력 Text}) =$

$$P(\text{you} | \text{정상 Mail}) \times P(\text{free} | \text{정상 Mail}) \times P(\text{lottery} | \text{정상 Mail}) \times P(\text{정상 Mail})$$

$P(\text{Spam Mail} | \text{입력 Text}) =$

$$P(\text{you} | \text{Spam Mail}) \times P(\text{free} | \text{Spam Mail}) \times P(\text{lottery} | \text{Spam Mail}) \times P(\text{Spam Mail})$$

$P(\text{정상 Mail}) = P(\text{Spam Mail}) = \text{총 Mail 6개 중 3개} = 0.5$

- 위 예제에서는  $P(\text{정상 Mail})$ 과  $P(\text{Spam Mail})$ 의 값은 같으므로, 두 식에서 두 개의 확률은 생략이 가능하다.

$P(\text{정상 Mail} | \text{입력 Text}) =$

$$P(\text{you} | \text{정상 Mail}) \times P(\text{free} | \text{정상 Mail}) \times P(\text{lottery} | \text{정상 Mail})$$

$P(\text{Spam Mail} | \text{입력 Text}) =$

$$P(\text{you} | \text{Spam Mail}) \times P(\text{free} | \text{Spam Mail}) \times P(\text{lottery} | \text{Spam Mail})$$

## Spam Detection (Cont.)

- $P(\text{you} \mid \text{정상 Mail})$ 을 구하는 방법은 정상 Mail에 등장한 모든 단어의 빈도 수의 총합을 분모로하고, 정상 Mail에서 you가 총 등장한 빈도의 수를 분자로 하는 것이다.
- 이 경우에는  $2/10 = 0.2$ 가 된다.
- 이와 같은 원리로 식을 전개하면 이와 같습니다.

$$P(\text{정상 Mail} \mid \text{입력 Text}) = 2/10 \times 2/10 \times 0/10 = 0$$

$$P(\text{Spam Mail} \mid \text{입력 Text}) = 2/10 \times 3/10 \times 2/10 = 0.012$$

- 결과적으로  $P(\text{정상 Mail} \mid \text{입력 Text}) < P(\text{Spam Mail} \mid \text{입력 Text})$ 이므로 입력 테스트 *you free lottery*는 Spam Mail로 분류된다.
- 혹시 Naïve Bayes 분류기에서는 각 단어에 대한 확률의 분모, 문자에 전부 숫자를 더해서 문자가 0이 되는 것을 방지하는 Laplace Smoothing을 사용하기도 한다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier)

## ■ Data Source

- Scikit-learn에서 제공하는 Twenty Newsgroups이라고 불리는 20개의 다른 주제를 가진 18,846개의 News Data
- 해당 Data는 이미 Train Data(News 11,314개)와 Test Data(News 7,532개)를 사전 분류해놓았음.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) (Cont.)

## ■ News Data에 대한 이해

- Data는 총 6개의 속성을 갖고 있다.
- 그 중에서 우리가 사용할 것은 해당 Data의 본문을 갖고 있는 **data** 속성과 해당 Data가 어떤 Category에 속하는지 0부터 19까지의 Label이 붙어있는 **target** 속성이다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) -News Data에 대한 이해

## ■ Train Data Download.

```
1 from sklearn.datasets import fetch_20newsgroups  
2 newsdata=fetch_20newsgroups(subset='train')  
3 print(newsdata.keys())
```

```
dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR'])
```

- 위의 코드 부분에 **subset** 부분에 *all*을 넣으면 모든 Data인 News 18,846 개를 Download할 수 있으며, *train*을 넣으면 Train Data를, *test*를 넣으면 Test Data를 Download 할 수 있다.
- **newsdata.keys()** 를 출력하여 해당 데이터의 속성을 알아보았다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) -News Data에 대한 이해 (Cont.)

## ■ Train Data Download.

```
1 from sklearn.datasets import fetch_20newsgroups  
2 newsdata=fetch_20newsgroups(subset='train')  
3 print(newsdata.keys())
```

```
dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR'])
```

- 위의 코드 부분에 **subset** 부분에 *all*을 넣으면 모든 Data인 News 18,846 개를 Download할 수 있으며, *train*을 넣으면 Train Data를, *test*를 넣으면 Test Data를 Download 할 수 있다.
- **newsdata.keys()**를 출력하여 해당 데이터의 속성을 알아보았다.
- 해당 데이터는 *data*, *filenames*, *target\_names*, *target*, *DESC*이라는 5개 속성의 데이터를 갖고 있다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) -News Data에 대한 이해 (Cont.)

- Train News의 개수를 확인해 본다.

```
1 print (len(newsdata.data), len(newsdata.filenames), #  
2     len(newsdata.target_names), len(newsdata.target))
```

11314 11314 20 11314

- 훈련용 News는 총 11,314개로 구성되어 있다.
- **newsdata.target\_names**는 이 News Data의 20개의 Category의 이름을 담고 있다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) -News Data에 대한 이해 (Cont.)

- 어떤 Category들로 구성되어있는지 확인해보도록 하자.

```
1 print(newsdata.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']
```

- **target**에는 총 0부터 19까지의 숫자가 들어가 있는데 첫 번째 훈련용 News의 경우에는 몇 번 Category인지 확인해 본다.

```
1 print(newsdata.target[0])
```

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) -News Data에 대한 이해 (Cont.)

- 첫 번째 훈련용 News는 Category 7번에 속한다고 Label이 붙어있다.
- 7번 Category의 Category 제목은 **rec.autos**이다.
- 결국, 첫 번째 훈련용 News는 **rec.autos** Category에 속한다는 것을 알 수 있다.

```
1 print(newsdata.target_names[7])
```

rec.autos

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) -News Data에 대한 이해 (Cont.)

- 첫 번째 훈련용 뉴스가 어떤 내용을 갖고 있는지 확인해보자.

```
1 print(newsdata.data[0])
```

From: lerxst@wam.umd.edu (where's my thing)  
Subject: WHAT car is this?  
Nntp-Posting-Host: rac3.wam.umd.edu  
Organization: University of Maryland, College Park  
Lines: 15

I was wondering if anyone out there could enlighten me on this car I saw the other day. It was a 2-door sports car, looked to be from the late 60s/early 70s. It was called a Bricklin. The doors were really small. In addition, the front bumper was separate from the rest of the body. This is all I know. If anyone can tell me a model name, engine specs, years of production, where this car is made, history, or whatever info you have on this funky looking car, please e-mail.

Thanks,

- IL

---- brought to you by your neighborhood Lerxst ----

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) –Naïve Bayes 분류

- Download 받은 훈련 Data에 대한 Preprocessing을 진행한다.
- 사용할 Data는 `newsdata.data`와 그에 대한 Category Label이 되어있는 `newsdata.target`이다.
- 여기서 Preprocessing을 하는 Data는 `newsdata.data`이다.
- 해당 Data는 Token化가 전혀 되어있지 않다.
- 따라서 Naïve Bayes 분류를 위해서는 Data를 BoW로 만들어야 한다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) –Naïve Bayes 분류 (Cont.)

- 여기서는 입력한 텍스트를 자동으로 BoW로 만드는 **CountVectorizer**를 사용한다.

```
1 from sklearn.feature_extraction.text import CountVectorizer  
2 dtmvector = CountVectorizer()  
3 X_train_dtm = dtmvector.fit_transform(newsdata.data)  
4 print(X_train_dtm.shape)
```

(11314, 130107)

- DTM이 완성되었다.
- 11,314는 훈련용 News의 개수이고 DTM 관점에서는 문서의 수가 됐다.
- 130,107은 전체 훈련 Data에 등장한 단어의 수를 의미한다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) –Naïve Bayes 분류 (Cont.)

- DTM 행렬 대신 TF-IDF 가중치를 적용한 TF-IDF 행렬을 입력으로 Text 분류를 수행하면, 성능의 개선을 얻을 수 있다.
- 주의할 점은 TF-IDF 행렬이 항상 DTM으로 수행했을 때보다 성능이 뛰어나지는 않다.
- Scikit-learn에서 제공하는 **TfidfVectorizer** 클래스를 사용하여 TF-IDF를 자동 계산하자.

```
1 from sklearn.feature_extraction.text import TfidfTransformer  
2 tfidf_transformer = TfidfTransformer()  
3 tfidfv = tfidf_transformer.fit_transform(X_train_dtm)  
4 print(X_train_dtm.shape)
```

(11314, 130107)

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) –Naïve Bayes 분류 (Cont.)

- TF-IDF 행렬이 만들어졌다.
- Scikit-learn은 Naïve Bayes Model을 지원하므로, 이를 그대로 사용하겠다.

```
1 from sklearn.naive_bayes import MultinomialNB # 다항분포 Naive Bayes Model  
2 mod = MultinomialNB()  
3 mod.fit(tfidf, newsdata.target)
```

MultinomialNB(alpha=1.0, class\_prior=None, fit\_prior=True)

- 모델의 입력으로 TF-IDF 행렬과 11,314개의 훈련 Data에 대한 Label이 적혀있는 **newsdata.target**이 들어간다.
- 이것은 각각 **x\_train**과 **y\_train**에 해당되는 Data들이다.
- 여기서 **alpha=1.0**은 Laplace Smoothing 이 적용되었음을 의미한다.

# News Data 분류하기(Classification of 20 News Group with Naïve Bayes Classifier) –Naïve Bayes 분류 (Cont.)

```
1 from sklearn.metrics import accuracy_score #정확도 계산을 위한 함수  
2 newsdata_test = fetch_20newsgroups(subset='test', shuffle=True) #테스트 데이터 갖고오기  
3 X_test_dtm = dtmvector.transform(newsdata_test.data) #테스트 데이터를 DTM으로 변환  
4 tfidfv_test = tfidf_transformer.transform(X_test_dtm) #DTM을 TF-IDF 행렬로 변환  
5  
6 predicted = mod.predict(tfidfv_test) #테스트 데이터에 대한 예측  
7 print("정확도:", accuracy_score(newsdata_test.target, predicted)) #예측값과 실제값 비교
```

정확도 : 0.7738980350504514

---

## 6. Naver Movie Review Sentiment Analysis



# Naver Movie Review Sentiment Analysis

- 사용할 Data는 Naver Movie Review Data이다.
- 총 200,000개 Review로 구성된 Data
- 구성
  - Movie Review에 대한 Text
  - Review가 긍정인 경우 1, 부정인 경우 0으로 표시한 Label

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해

- Downloads : <https://github.com/e9t/nsmc/>
- 훈련 Data에 해당하는 ratings\_train.txt와 Test Data에 해당하는 **ratings\_test.txt**를 Download한다.

```
1 import pandas as pd  
2 train_data= pd.read_table('./ratings_train.txt')  
3 test_data= pd.read_table('./ratings_test.txt')
```

```
1 print(len(train_data)) # 리뷰 개수 출력
```

150000

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- train\_data는 어떤 형태인지 상위 5개만 출력한다.

```
1 train_data[:5] # 상위 5개 출력
```

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

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

### ■ test\_data의 개수를 알아보자.

```
1 print(len(test_data))
```

50000

### ■ test\_data의 상위 5개를 출력해보자.

```
1 test_data[:5]
```

	<b>id</b>	<b>document</b>	<b>label</b>
0	6270596	굳ㅋ	1
1	9274899	GDNTOPCLASSINTHECLUB	0
2	8544678	뭐야 이 평점들은.... 나쁘진 않지만 10점 짜리는 더더욱 아니잖아	0
3	6825595	지루하지는 않은데 완전 막장임... 돈주고 보기에는....	0
4	6723715	3D만 아니었어도 별 다섯 개 줬을텐데.. 왜 3D로 나와서 제 심기를 불편하게 하죠??	0

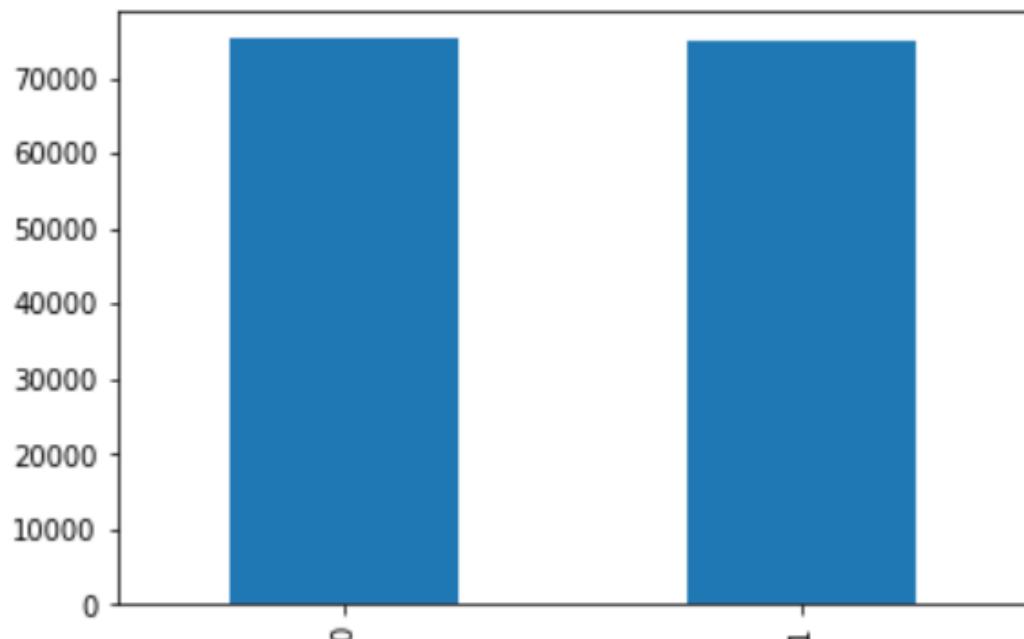
# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- **train\_data**에서 해당 리뷰의 긍, 부정 유무가 기재되어있는 Label 값의 분포를 보자.

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 train_data['label'].value_counts().plot(kind='bar')
```

<matplotlib.axes.\_subplots.AxesSubplot at 0x7f00e50a75c0>



# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- train\_data에서 해당 리뷰의 정확하게 몇 개인지 확인해 보자.

```
1 print(train_data.groupby('label').size().reset_index(name='count'))
```

	label	count
0	0	75173
1	1	74827

- Label이 0인 Review가 조금 더 많은 것을 알 수 있다.

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- Review 중에 Null 값을 가진 Sample이 있는지 Pandas의 `isnull().values.any()`로 확인한다.

```
1 print(train_data.isnull().values.any())
```

True

- **True**가 나왔다면 Data 중에 **Null** 값을 가진 Sample이 존재한다는 의미이다.
- 어떤 열에 존재하는지 확인해 보자.

```
1 print(train_data.isnull().sum())
```

id	0
document	5
label	0
dtype:	int64

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- Review가 적혀있는 document 열에서 Null 값을 가진 Sample이 총 5개가 존재한다고 확인했다.
- 그렇다면 document 열에서 Null 값이 존재한다는 것을 조건으로 Null 값을 가진 Sample이 어느 Index의 위치에 존재하는지 확인해 보자.

```
1 train_data.loc[train_data.document.isnull()]
```

	<b>id</b>	<b>document</b>	<b>label</b>
25857	2172111	NaN	1
55737	6369843	NaN	1
110014	1034280	NaN	0
126782	5942978	NaN	0
140721	1034283	NaN	0

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- Null 값을 가진 Sample을 제거한다.

```
1 train_data=train_data.dropna(how='any') # Null 값이 존재하는 행 제거  
2 print(train_data.isnull().values.any()) # Null 값이 존재하는지 확인
```

False

- Null 값을 가진 Sample이 제거되었다.
- 다시 Sample의 개수를 출력하여 5개의 Sample이 제거되었는지 확인한다.

```
1 print(len(train_data))
```

149995

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- Data의 Preprocessing을 수행한다.
- train\_data와 test\_data에서 온점(.)이나 ?와 같은 각종 특수문자가 사용된 것을 확인했다면 정규식을 사용해서 처리해야 한다.
- train\_data로부터 한글만 남기고 특수문자를 제거하자.
- 우선 한글을 범위 지정할 수 있는 정규 표현식을 찾아보자.
- 자음과 모음에 대한 범위를 지정하면...
- 일반적으로 자음의 범위는 ㄱ ~ ㅎ, 모음의 범위는 ㅏ ~ ㅣ와 같이 지정할 수 있다.
- 해당 범위 내에 어떤 자음과 모음이 속하는지 알고 싶다면 아래의 링크를 참고해 보자.
- <https://www.unicode.org/charts/PDF/U3130.pdf>

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- ㄱ ~ ㅎ: 3131 ~ 314E  
ㅏ ~ ㅣ: 314F ~ 3163
- 완성형 한글의 범위는 가~ 흥과 같이 사용한다.
- 해당 범위 내에 포함된 음절들은 아래의 링크에서 확인할 수 있다.  
<https://www.unicode.org/charts/PDF/UAC00.pdf>
- 위의 범위 지정을 모두 반영하여 train\_data에 한글과 공백을 제외하고 모두 제거하는 정규 표현식을 수행해보자.

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

```
1 train_data['document'] = train_data['document'].str.replace("[^ㄱ-ㅎㅏ-ㅣ가-힣 ]","")
2 # 한글과 공백을 제외하고 모두 제거
3 train_data[:5]
```

	<b>id</b>	<b>document</b>	<b>label</b>
0	9976970	아 더 빙 진짜 짜증나네요 목소리	0
1	3819312	홈포스터보고 초딩영화줄오버연기조차 가볍지 않구나	1
2	10265843	너무재밌었다그래서보는것을추천한다	0
3	9045019	교도소 이야기구먼 솔직히 재미는 없다평점 조정	0
4	6483659	사이몬페그의 익살스런 연기가 돋보였던 영화스파이더맨에서 늙어보이기만 했던 커스틴 던...	1

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- 이제 불용어 제거를 한다.
- 불용어는 개별로 정의해야 한다.

```
1 stopwords=['의','가','이','은','들','는','좀','잘','걍','과','도','를','으로','■  
2         '자','에','와','한','하다']
```

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- 토큰화를 위한 형태소 분석기는 KoNLPy의 *Okt*를 사용한다.
- 한국어을 토큰화할 때는 영어처럼 띄어쓰기 기준으로 토큰화를 하는 것이 아니라, 주로 형태소 분석기를 사용한다.
- **stem=True**를 사용하면 일정 수준의 정규화를 수행한다.
- 이제 **train\_data**에 형태소 분석기를 사용하여 토큰화를 하면서 불용어를 제거하여 **x\_train**에 저장해보자.

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

```
1 X_train=[]  
2 for sentence in train_data['document']:  
3     temp_X = []  
4     temp_X=okt.morphs(sentence, stem=True) # 토큰화  
5     temp_X=[word for word in temp_X if not word in stopwords] # 불용어 제거  
6     X_train.append(temp_X)
```

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- 상위 3개의 Sample만 출력하여 결과를 확인해본다.

```
1 print(X_train[:3])
```

```
[['아', '더빙', '진짜', '짜증나다', '목소리'], ['흠', '포스터', '보고', '초딩', '영화', '줄', '오버',
'연기', '조차', '가볍다', '않다'], ['너', '무재', '밀었', '다그', '래서', '보다', '추천', '다']]
```

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- Test Data인 **test\_data**에 방금 진행했던 과정들을 동일하게 진행한다.

```
1 test_data=test_data.dropna(how='any') # Null 값 제거
2 test_data['document'] = test_data['document'].str.replace("[^ㄱ-ㅎㅏ-ㅣ가-힣 ]","") # 정규 표현식
3
4 X_test=[]
5 for sentence in test_data['document']:
6     temp_X = []
7     temp_X=okt.morphs(sentence, stem=True) # 토큰화
8     temp_X=[word for word in temp_X if not word in stopwords] # 불용어 제거
9     X_test.append(temp_X)
```

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- 훈련 Data **x\_train**과 Test Data **x\_test**에 대해서 Integer Encoding을 수행한다.

```
1 from keras.preprocessing.text import Tokenizer  
2 max_words = 35000  
3 tokenizer = Tokenizer(num_words=max_words) # 상위 35,000개의 단어만 보존  
4 tokenizer.fit_on_texts(X_train)  
5 X_train = tokenizer.texts_to_sequences(X_train)  
6 X_test = tokenizer.texts_to_sequences(X_test)
```

Using TensorFlow backend.

# Naver Movie Review Sentiment Analysis

## -Naver Movie Review Data에 대한 이해 (Cont.)

- Integer Encoding이 진행되었는지 확인하고자 **x\_train**에 대해서 상위 3 개의 Sample만 출력한다.

```
1 print(X_train[:3])
```

```
[[50, 457, 16, 260, 660], [919, 459, 41, 599, 1, 214, 1455, 24, 965, 676, 19], [386, 2452, 25024, 231  
9, 5676, 2, 221, 9]]
```

- 각 Sample은 단어 대신 단어에 대한 Index가 부여된 것을 확인할 수 있다.

# Naver Movie Review Sentiment Analysis

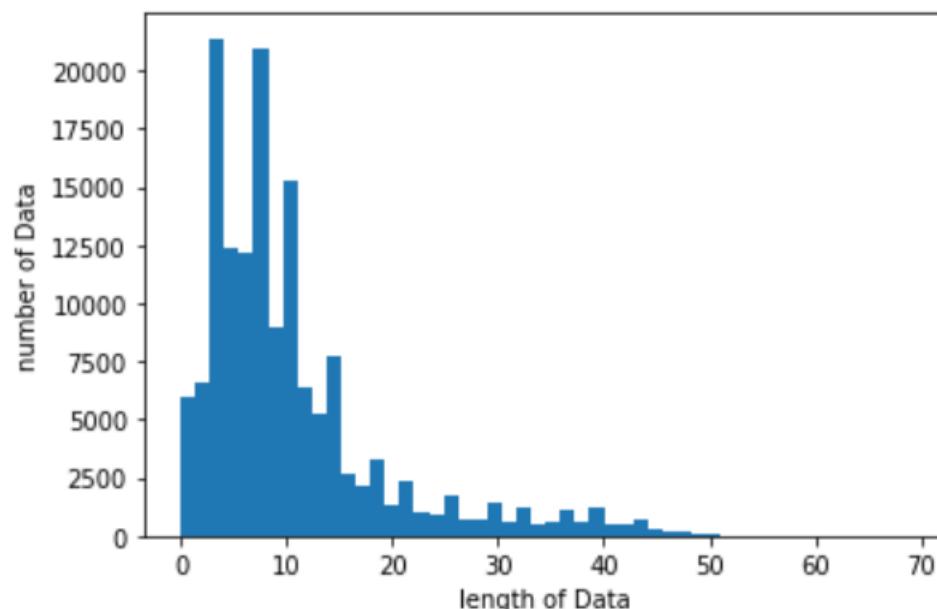
## -Naver Movie Review Data에 대한 이해 (Cont.)

- 전체 Data에서 가장 길이가 긴 Review와 전체 Data의 길이 분포를 알아본다.

```
1 print('리뷰의 최대 길이 :',max(len(l) for l in X_train))
2 print('리뷰의 평균 길이 :',sum(map(len, X_train))/len(X_train))
3 plt.hist([len(s) for s in X_train], bins=50)
4 plt.xlabel('length of Data')
5 plt.ylabel('number of Data')
6 plt.show()
```

리뷰의 최대 길이 : 69

리뷰의 평균 길이 : 10.64700156671889



# Naver Movie Review Sentiment Analysis

## -LSTM으로 네이버 영화 리뷰 감성 분류하기

1. Model을 만들기 전에 **train\_data**와 **test\_data**에 존재하는 Label을 따로 **y\_train**과 **y\_test**에 저장한다.

```
1 y_train=train_data['label']
2 y_test=test_data['label']
```

# Naver Movie Review Sentiment Analysis

## -LSTM으로 네이버 영화 리뷰 감성 분류하기 (Cont.)

### 2. 필요한 Package들을 가져온다.

```
1 from keras.layers import Embedding, Dense, LSTM  
2 from keras.models import Sequential  
3 from keras.preprocessing.sequence import pad_sequences
```

### 3. Model처리할 수 있도록 **x\_train**과 **x\_test**의 모든 Sample의 길이를 동일하게 했다. ● 여기서는 길이를 30으로 정했다.

```
1 max_len=30  
2 # 전체 데이터의 길이는 30으로 맞춘다.  
3 X_train = pad_sequences(X_train, maxlen=max_len)  
4 X_test = pad_sequences(X_test, maxlen=max_len)
```

# Naver Movie Review Sentiment Analysis

## -LSTM으로 네이버 영화 리뷰 감성 분류하기 (Cont.)

### 4. Model을 생성한다.

```
1 model = Sequential()
2 model.add(Embedding(max_words, 100))
3 model.add(LSTM(128))
4 model.add(Dense(1, activation='sigmoid'))
5
6 model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
7 history = model.fit(X_train, y_train, epochs=4, batch_size=60, validation_split=0.2)
```

- Embedding Vector의 차원은 100으로 정했고, Review 분류를 위해서 LSTM을 사용한다.

# Naver Movie Review Sentiment Analysis

## -LSTM으로 네이버 영화 리뷰 감성 분류하기 (Cont.)

- Epoch는 총 4번을 수행했다.
- 훈련 Data 중 20%를 검증 Data로 사용하면서 정확도를 확인한다.

Train on 119996 samples, validate on 29999 samples

Epoch 1/4

```
119996/119996 [=====] - 188s 2ms/step - loss: 0.3903 - acc: 0.8217 - val_loss: 0.3579 - val_acc: 0.8381
```

Epoch 2/4

```
119996/119996 [=====] - 187s 2ms/step - loss: 0.3292 - acc: 0.8556 - val_loss: 0.3328 - val_acc: 0.8535
```

Epoch 3/4

```
119996/119996 [=====] - 182s 2ms/step - loss: 0.3020 - acc: 0.8699 - val_loss: 0.3299 - val_acc: 0.8568
```

Epoch 4/4

```
119996/119996 [=====] - 184s 2ms/step - loss: 0.2792 - acc: 0.8830 - val_loss: 0.3303 - val_acc: 0.8552
```

# Naver Movie Review Sentiment Analysis

## -LSTM으로 네이버 영화 리뷰 감성 분류하기 (Cont.)

5. 훈련이 다 되었다면 이제 Test Data에 대해서 정확도를 측정한다.

```
1 print("테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
```

```
49997/49997 [=====] - 13s 259us/step
```

테스트 정확도: 0.8521

- Null 값을 가진 Sample이 제거되어 Test Data의 Sample은 49,997개가 존재한다.
- Test Data에서 85%의 정확도를 얻었다.



---

### III. Tagging Task

# 1. Tagging Task using Keras



# Tagging Task

- Tagging Task
  - 자연어 처리 분야에서 각 단어가 어떤 유형에 속해있는지를 알아내는 작업
- 개체명 인식 (Named Entity Recognition)
  - 각 단어의 유형이 사람, 장소, 단체 등 어떤 유형인지를 알아내는 작업
- 품사 태깅(Part-of-Speech Tagging)
  - 각 단어의 품사가 명사, 동사, 형용사 인지를 알아내는 작업
- Keras를 이용해서 ANN을 이용한 개체명 인식기와 품사 Tagger를 만드는 간단한 Mini Project를 다루려고 한다.

# Tagging Task using Keras

- Keras를 사용하여 개체명 인식기와 품사 Tagger를 만든다.
- 두 작업의 공통점
  - RNN의 다-대-다(Many-to-Many) 작업
  - 앞, 뒤 시점의 입력을 모두 참고하는 양방향 RNN(Bidirectional RNN)을 사용

# 훈련 데이터에 대한 이해

- Tagging 작업은 Supervised Learning이다.

- X

- Tagging을 해야 하는 단어 Data

- y

- Label에 해당되는 Tagging 정보 Data

- X\_train

- X에 대한 훈련 Data

- X\_test

- X에 대한 Test Data

- y\_train

- y에 대한 훈련 Data

- y\_test

- y에 대한 Test Data

## 훈련 데이터에 대한 이해 (Cont.)

- X와 y Data의 쌍(pair)은 병렬 구조를 가진다.
- X와 y의 각 Data의 길이는 같다.
- X\_train과 y\_train의 Data 중 4개의 Data만 확인해 본다면 구조는 다음 테이블과 같다.

-	X_train	y_train	길이
0	['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb']	['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O']	8
1	['peter', 'blackburn']	['B-PER', 'I-PER']	2
2	['brussels', '1996-08-22']	['B-LOC', 'O']	2
3	['The', 'European', 'Commission']	['O', 'B-ORG', 'I-ORG']	3

```
graph TD; X0["X_train[0]"] -- red --> Y0["y_train[0]"]; X1["X_train[1]"] -- blue --> Y1["y_train[1]"]; X2["X_train[2]"] -- green --> Y2["y_train[2]"]; X3["X_train[3]"] -- red --> Y3["y_train[3]"]; X4["X_train[4]"] -- blue --> Y4["y_train[4]"]; X5["X_train[5]"] -- green --> Y5["y_train[5]"];
```

# Sequence Labeling

## ■ Sequence Labeling Task

- 입력 Sequence  $X = [x_1, x_2, x_3, \dots, x_n]$ 에 대하여 Label Sequence  $y = [y_1, y_2, y_3, \dots, y_n]$ 를 각각 부여하는 작업

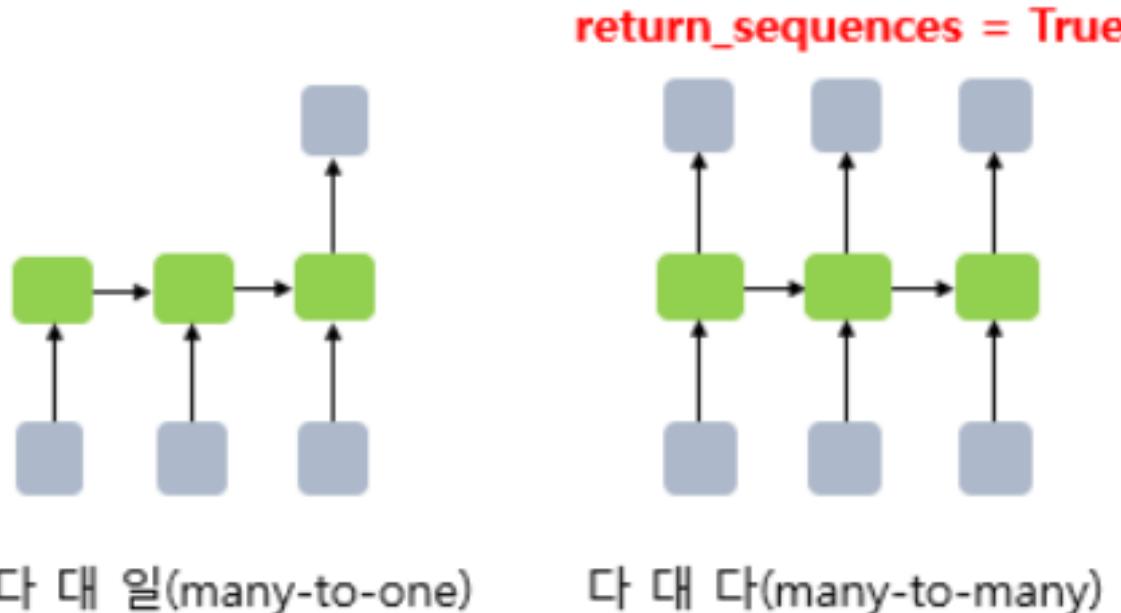
■ Tagging 작업은 대표적인 Sequence Labeling 작업이다.

# Bidirectional LSTM

```
model.add(  
    Bidirectional(  
        LSTM(  
            hidden_size, return_sequences=True) ))
```

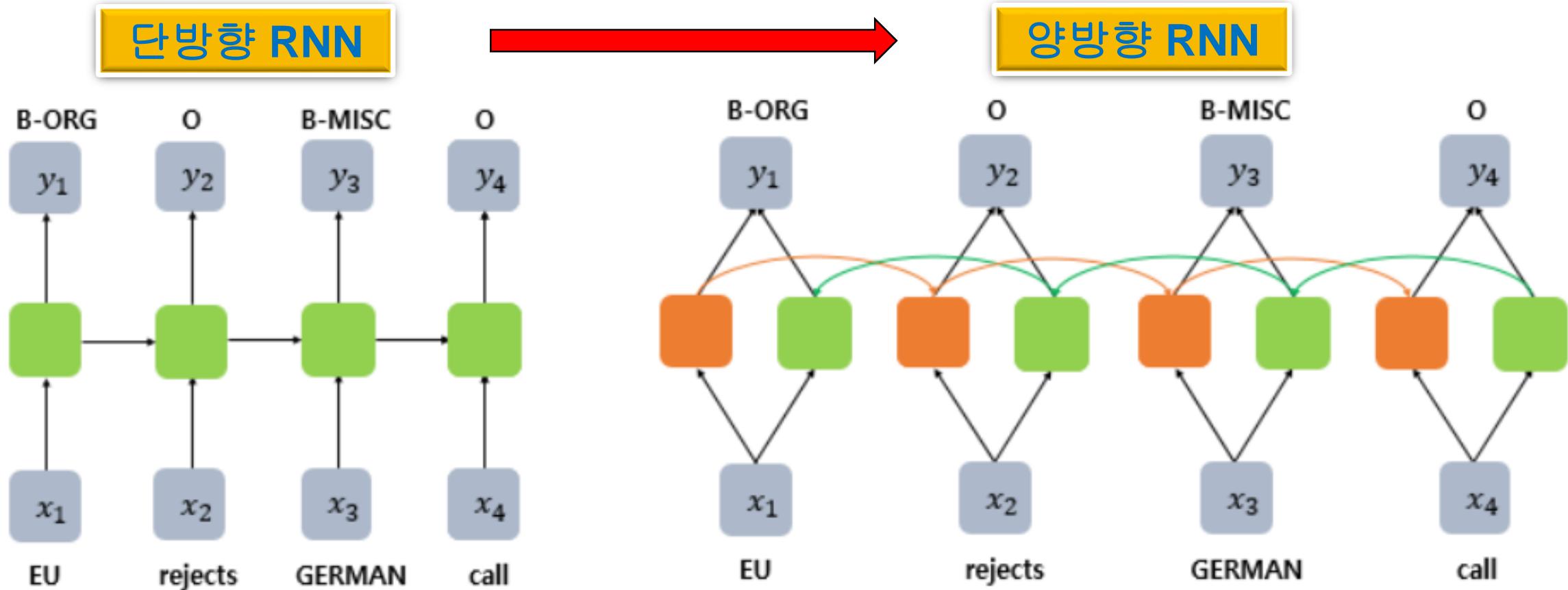
- Text Classification → 단방향 LSTM
- Tagging Task → 양방향 LSTM
  - 이전 시점의 단어 정보 뿐만 아니라, 다음 시점의 단어 정보도 참고하기 위함이다.
  - 양방향은 기존의 단방향 LSTM()을 Bidirectional() 안에 넣으면 된다.

# RNN의 다-대-다(Many-to-Many)문제



- RNN의 은닉층은 모든 시점에 대해서 은닉 상태값을 출력할 수 있고, 마지막 시점에 대해서만 은닉 상태값을 출력할 수 있다.
- `return_sequences`를 `True`인지 `False`에 따라 구별할 수 있다.
- Many-to-Many 문제는 `return_sequences=True`를 설정.

# RNN의 다-대-다(Many-to-Many)문제 (Cont.)



## 2. Named Entity Recognition



# 개체명 인식(Named Entity Recognition)이란?

- 이름을 가진 개체(named entity)를 인식하겠다는 것을 의미
- 어떤 이름을 의미하는 단어를 보고는 그 단어가 어떤 유형인지를 인식하는 것을 말한다.
- 예)

**도연이는 2018년에 골드만삭스에 입사했다.**

-사람(person), 조직(organization), 시간(time)에 대해 개체명 인식

- 도연 → Person
- 2018년 → Time
- 골드만삭스 → Organization

# Named Entity Recognition using NLTK

- NLTK에서 개체명 인식기(NER chunker) 지원
- NLTK를 사용해서 개체명 인식을 수행할 수 있다.

```
1 import nltk  
2 nltk.download('maxent_ne_chunker')  
3 nltk.download('words')
```

```
[nltk_data] Downloading package maxent_ne_chunker to  
[nltk_data]      /home/instructor/nltk_data...  
[nltk_data]  Unzipping chunkers/maxent_ne_chunker.zip.  
[nltk_data] Downloading package words to /home/instructor/nltk_data...  
[nltk_data]  Unzipping corpora/words.zip.
```

True

```
1 from nltk import word_tokenize, pos_tag, ne_chunk  
2 sentence = "James is working at Disney in London"  
3 sentence = pos_tag(word_tokenize(sentence))  
4 print(sentence) # 토큰화와 품사 태깅을 동시에 수행
```

```
[('James', 'NNP'), ('is', 'VBZ'), ('working', 'VBG'), ('at', 'IN'), ('Disney', 'NNP'), ('in', 'IN'),  
('London', 'NNP')]
```

# Named Entity Recognition using NLTK (Cont.)

```
1 sentence=ne_chunk(sentence)
2 print(sentence) # 개체명 인식
```

```
(S
  (PERSON James/NNP)
  is/VBZ
  working/VBG
  at/IN
  (ORGANIZATION Disney/NNP)
  in/IN
  (GPE London/NNP))
```

- **ne\_chunk** 개체명을 Tagging

- 위의 결과에서 James는 **PERSON**(사람), Disney는 **ORGANIZATION**(조직), London은 **GPE**(위치)라고 정상적으로 개체명 인식이 수행된 것을 볼 수 있다.



### **3. Named Entity Recognition using Bi-LSTM**



# BIO 표현

## ■ IOB(또는 BIO) 방법

- 개체명 인식에서 Corpus로부터 개체명을 인식하기 위한 방법
- B(Begin) : 개체명이 시작되는 부분
- I(Inside) : 개체명의 내부 부분을 의미
- O(Outside) : 개체명이 아닌 부분을 의미.

## ■ 예)

- 영화에 대한 Corpus 중에서 영화 제목에 대한 개체명을 뽑아내고 싶다고 가정.

해 리 포 터 보 러 가 자  
B I I I O O O O

개체명

## BIO 표현 (Cont.)

- 해 B - movie
- 리 I - movie
- 포 I - movie
- 터 I - movie
- 보 O
- 러 O
- 메 B - theater
- 가 I - theater
- 박 I - theater
- 스 I - theater
- 가 O
- 자 O

개체명

개체명

# Bi-directional LSTM으로 개체명 인식기 만들기

- 실습을 통해 양방향 LSTM을 이용한 개체명 인식에 대해서 더 자세히 알아보도록 한다.
- CONLL2003은 개체명 인식을 위한 전통적인 English DataSet이다.
- Download Link : <https://raw.githubusercontent.com/Franck-Dernoncourt/NeuroNER/master/neuroner/data/conll2003/en/train.txt>

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

## ■ train.txt

```
-DOCSTART- -X- -X- 0
EU NNP B-NP B-ORG
rejects VBZ B-VP 0
German JJ B-NP B-MISC
call NN I-NP 0
to TO B-VP 0
boycott VB I-VP 0
British JJ B-NP B-MISC
lamb NN I-NP 0
. . 0 0
Peter NNP B-NP B-PER
Blackburn NNP I-NP I-PER
BRUSSELS NNP B-NP B-LOC
1996-08-22 CD I-NP 0
```

품사 Tagging VBZ(단수 동산 현재형)

품사 Tagging NNP(고유명사단수형)

[단어][품사 Tagging][Chunk Tagging][개체명 Tagging]

각 품사에 대한 약어 설명  
[https://www.ling.upenn.edu/courses/Fall\\_2003/ling001/penn\\_treebank\\_pos.html](https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html)

개체명 Tagging ORG(Organization)

개체명 Tagging PER(Person)

개체명 Tagging LOC(Location)

개체명 Tagging MISC(Miscellaneous)

공백 라인은 새로운 문장이 시작된다는 의미

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 from collections import Counter  
2 vocab=Counter()  
3 import re
```

- 훈련 Data의 단어의 빈도수를 세기 위해서 **Counter**, 데이터를 정제하기 위해서 **re**가 필요하다.

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 f = open('./train.txt', 'r')
2
3 sentences = []
4 sentence = []
5 ner_set = set()
6
7 for line in f:
8     if len(line)==0 or line.startswith('-DOCSTART') or line[0]=="\n":
9         if len(sentence) > 0:
10             sentences.append(sentence)
11             sentence=[]
12         continue
13     splits = line.split(' ')
14     # 공백을 기준으로 속성을 구분한다.
15     splits[-1] = re.sub(r'\n', '', splits[-1])
16     # 개체명 태깅 뒤에 붙어있는 줄바꿈 표시 \n을 제거한다.
17     word=splits[0].lower()
18     # 단어들은 소문자로 바꿔서 저장한다. 단어의 수를 줄이기 위해서이다.
19     vocab[word]=vocab[word]+1
20     # 단어마다 빈도 수가 몇 인지 기록한다.
21     sentence.append([word, splits[-1]])
22     # 단어와 개체명 태깅만 기록한다.
23     ner_set.add(splits[-1])
24     # set에다가 개체명 태깅을 집어 넣는다. 중복은 허용되지 않으므로
25     # 나중에 개체명 태깅이 어떤 종류가 있는지 확인할 수 있다.
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 이제 단어와 개체명 Tagging만이 남게 되며, 문장의 구분이 유지된다.

```
1 sentences[:3]
```

```
[[[['eu', 'B-ORG'],
  ['rejects', 'O'],
  ['german', 'B-MISC'],
  ['call', 'O'],
  ['to', 'O'],
  ['boycott', 'O'],
  ['british', 'B-MISC'],
  ['lamb', 'O'],
  ['. ', 'O']],
 [[['peter', 'B-PER'], ['blackburn', 'I-PER']],
  [['brussels', 'B-LOC'], ['1996-08-22', 'O']]]]
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 앞의 코드 중 **vocab[word]=vocab[word]+1** 해당 부분을 통해 현재 vocab은 각 단어에 대한 빈도수가 기록되어 있는 단어 집합이 되었다.

In [4]: 1 vocab

```
Out[4]: Counter({'eu': 24,
                  'rejects': 1,
                  'german': 101,
                  'call': 38,
                  'to': 3424,
                  'boycott': 5,
                  'british': 96,
                  'lamb': 3,
                  '.': 7374,
                  'peter': 31,
                  'blackburn': 12,
                  'brussels': 33,
                  '1996-08-22': 125,
                  'the': 8390,
                  'european': 94,
                  'commission': 67,
                  'said': 1849,
                  'on': 2092,
                  'thursday': 292,
                  'it': 762}
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- vocab이 위와 같은 데이터의 형식을 갖고 있기 때문에, vocab의 길이를 측정하면 단어의 개수를 알 수 있다.

```
1 len(vocab)
```

21009

- 개체명 Tagging의 종류를 알기 위해서 **ner\_set**에다가 중복은 허용하지 않고, 개체명 Tagging을 저장했다.
- 훈련 데이터에 등장하는 개체명 Tagging의 수는 아래와 같다.

```
1 print(ner_set)
```

{'B-MISC', 'B-LOC', 'B-ORG', 'I-ORG', 'B-PER', 'I-PER', 'I-MISC', 'O', 'I-LOC'}

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- vocab 안의 단어들을 빈도수 순으로 정렬해 본다.

```
In [7]: 1 vocab_sorted=sorted(vocab.items(), key=lambda x:x[1], reverse=True)
2 # vocab을 빈도수 순으로 정렬한다.
3 vocab_sorted
4 # 출력
```

```
Out[7]: [('the', 8390),
          ('.', 7374),
          (',', 7290),
          ('of', 3815),
          ('in', 3621),
          ('to', 3424),
          ('a', 3199),
          ('and', 2872),
          ('(', 2861),
          (')', 2861),
          ('"', 2178),
          ('on', 2092),
          ('said', 1849),
          ("'s", 1566),
          ('for', 1465),
          ('I', 1421),
          ('-', 1243),
          ('at', 1146),
          ('was', 1095),
          ('?', 972)]
```

## Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 등장 빈도가 1인 데이터도 다수 보인다.
- 전체 Data에서 단어의 등장 빈도수가 5이하인 경우는 Data에서 배제시키고, 모르는 단어. OOV(Out-of-Vocabulary)로 간주한다.
- 이제 **vocab\_sorted**로 부터 또 다시 새로운 단어 집합을 만든다.
- 이번에 만드는 단어 집합은 등장 빈도수 순으로 Index를 부여한다.
- 그리고 이 때, 빈도수가 5이하인 단어들을 배제시킨다.

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
In [32]: 1 word_to_index = {w: i + 2 for i, (w, n) in enumerate(vocab_sorted) if n > 5}
2 word_to_index['PAD'] = 0 # 패딩을 위해 인덱스 0 할당
3 word_to_index['OOV'] = 1 # 모르는 단어를 위해 인덱스 1 할당
4 word_to_index # 출력
```

```
Out[32]: {'the': 2,
'.': 3,
',': 4,
'of': 5,
'in': 6,
'to': 7,
'a': 8,
'and': 9,
'(': 10,
')': 11,
'"': 12,
'on': 13,
'said': 14,
"'s": 15,
'for': 16,
'1': 17,
'-': 18,
'at': 19,
'was': 20,
'2': 21}
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 해당 단어 집합은 단어의 등장 빈도 순위에 따라서 Index를 부여했다.
- **word\_to\_index**의 Index 0에 'PAD'와 Index 1에 'OOV'도 추가된 상태이다.
- Index 2부터는 등장 빈도수가 가장 높은 단어 순서대로 부여되는데, 등장 빈도수가 가장 높은 the의 경우에는 Index 2에 할당된다.
- 해당 단어 집합의 크기를 확인해 보자.

```
1 print(len(word_to_index))
```

3939

- 약 21,000개에 달했던 단어 집합이 빈도수 5이하인 단어들을 배제시키자, 3939개의 단어만을 가진 단어 집합으로 단어의 개수가 대폭 줄어든 것을 확인할 수 있다.

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- **word\_to\_index**를 통해 단어를 입력하면, Index를 리턴받을 수 있다.

```
1 word_to_index['the']
```

2

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 개체명 Tagging에 Index를 부여하고, 입력받은 개체명 Tagging에 대해서 Index를 return하는 **ner\_to\_index**를 만들어 보자.

```
1 ner_to_index={}
2 ner_to_index['PAD'] = 0
3 i=1
4 for ner in ner_set:
5     ner_to_index[ner]=i
6     i=i+1
7 print(ner_to_index)
```

```
{'PAD': 0, 'B-MISC': 1, 'B-LOC': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-PER': 5, 'I-PER': 6, 'I-MISC': 7, 'O': 8, 'I-LOC': 9}
```

- 이제 **ner\_to\_index**에다가 개체명 Tagging을 입력하면 Index를 return받을 수 있다.

```
1 ner_to_index['I-PER']
```

## Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 모든 훈련 Data를 담고 있는 `sentences`로 부터 `word_to_index`와 `ner_to_index`를 통해 모든 훈련 데이터를 숫자로 변경한다.
- 우선 `word_to_index`를 사용하여 단어에 대해서 훈련 데이터인 `data_x`를 만든다.

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

In [13]: ►

```
1 data_X = []
2
3 for s in sentences:
4     temp_X = []
5     for w, label in s:
6         try:
7             temp_X.append(word_to_index.get(w, 1))
8         except KeyError:
9             temp_X.append(word_to_index['OOV'])
10
11     data_X.append(temp_X)
12 print(data_X)
```

```
[ [989, 1, 205, 629, 7, 1, 216, 1, 3], [774, 1872], [726, 150], [2, 219, 334, 14, 13, 70, 28, 1, 24,
205, 1, 7, 2404, 7, 1, 216, 1, 406, 3382, 2009, 519, 1745, 1873, 648, 309, 41, 1, 7, 1632, 3], [124,
15, 2991, 7, 2, 219, 300, 15, 2660, 801, 1, 1, 14, 13, 75, 2404, 276, 914, 1, 27, 539, 127, 125, 13
7, 406, 2, 2405, 1, 20, 1, 3], [12, 69, 185, 213, 357, 220, 567, 1, 170, 69, 185, 213, 859, 220, 338
3, 16, 28, 4, 12, 2, 334, 15, 416, 187, 1, 594, 2010, 1, 90, 8, 188, 3384, 3], [26, 14, 683, 2405, 2
181, 20, 2661, 9, 141, 28, 20, 365, 25, 775, 20, 990, 28, 276, 41, 580, 29, 2, 219, 300, 3], [26, 1
4, 8, 2182, 66, 233, 29, 989, 1129, 2406, 1, 3385, 7, 595, 1632, 1, 4, 1, 9, 1, 1, 27, 2, 581, 9, 26
62, 1029, 3386, 20, 8, 3387, 2992, 9, 1, 1301, 7, 1, 581, 630, 3], [3385, 1544, 1, 1462, 42, 744, 2
7, 137, 9, 146, 25, 104, 1, 860, 1632, 126, 745, 1, 1, 1, 10, 2011, 11, 54, 1745, 1873, 648, 3], [3
7, 3385, 438, 7, 2993, 35, 2182, 42, 2, 989, 15, 2407, 2660, 801, 4, 1, 2662, 630, 138, 4, 1874, 14
1, 567, 775, 20, 1, 32, 100, 20, 143, 8, 1633, 1229, 7, 581, 630, 3], [1545, 1129, 103, 1, 371, 1, 3
4 024 021 0205 10 20 000 1100 1E40 100 020 5 1 1 100 10 1740 1 2 101 101 114]
```

- 위에 출력된 결과에서 중략 앞의 세 개의 list는 맨 처음에 출력했던 **sentences [:3]**의 세 개의 문장에 해당된다.

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 정상적으로 변환이 되었는지 보기 위해서 첫 번째 Sample에 대해서만 기존의 단어 Sequence를 출력해 본다.

```
1 index_to_word={}
2 for key, value in word_to_index.items(): # 인덱스를 단어로 바꾸기 위해 index_to_word를 생성
3     index_to_word[value] = key
4
5
6 temp = []
7 for index in data_X[0]: # 첫번째 샘플 안의 인덱스들에 대해서
8     temp.append(index_to_word[index]) # 다시 단어로 변환
9
10 print(sentences[0])
11 print(temp)
```

```
[['eu', 'B-ORG'], ['rejects', '0'], ['german', 'B-MISC'], ['call', '0'], ['to', '0'], ['boycott', '0'], ['british', 'B-MISC'], ['lamb', '0'], ['.', '0']]
['eu', 'OOV', 'german', 'call', 'to', 'OOV', 'british', 'OOV', '.']
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 이제 모든 훈련 Data를 담고 있는 **sentences**에서 개체명 Tagging에 해당되는 부분을 모아 **data\_y**에 저장하는 작업을 수행한다.

```
1 data_y = []
2
3 for s in sentences:
4     temp_y = []
5     for w, label in s:
6         temp_y.append(ner_to_index.get(label))
7
8     data_y.append(temp_y)
```

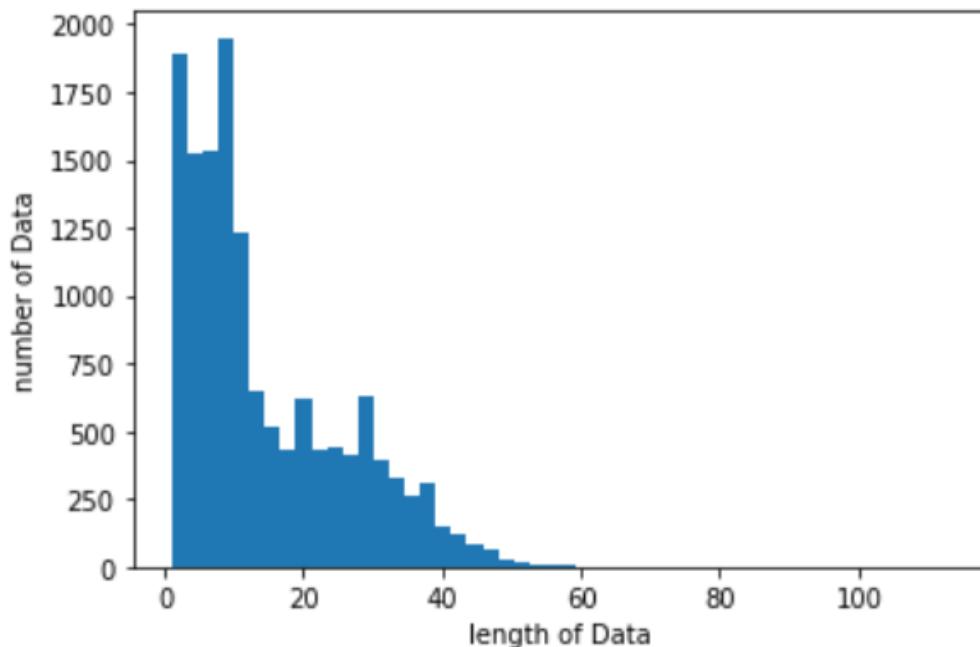
```
1 print(data_X[:4]) # X 데이터 4개만 출력
2 print(data_y[:4]) # y 데이터 4개만 출력
```

```
[[989, 1, 205, 629, 7, 1, 216, 1, 3], [774, 1872], [726, 150], [2, 219, 334, 14, 13, 70, 28, 1, 24, 2
05, 1, 7, 2404, 7, 1, 216, 1, 406, 3382, 2009, 519, 1745, 1873, 648, 309, 41, 1, 7, 1632, 3]]
[[3, 8, 1, 8, 8, 8, 1, 8, 8], [5, 6], [2, 8], [8, 3, 4, 8, 8, 8, 8, 8, 1, 8, 8, 8, 8, 1, 8, 8, 8, 8, 8, 8]]
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 각 샘플의 길이가 대체적으로 어떻게 되는지 시각화해 보자.

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3
4 plt.hist([len(s) for s in data_X], bins=50)
5 plt.xlabel('length of Data')
6 plt.ylabel('number of Data')
7 plt.show()
```



# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 양방향 LSTM Model에 손쉽게 Data를 입력으로 사용하기 위해서, 여기서는 모든 Sample의 길이를 동일하게 맞추도록 한다.
- 이에 따라 가장 길이가 긴 Sample의 길이를 우선 구해보자.

```
1 print(max([len(l) for l in data_X])) # 전체 데이터에서 길이가 가장 긴 샘플의 길이 출력  
2 print(max([len(l) for l in data_y])) # 전체 데이터에서 길이가 가장 긴 샘플의 길이 출력
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 가장 길이가 긴 Sample의 길이가 113이지만, 대부분의 Data가 40 ~ 60에 편중되어 있어서, 임의의 숫자 70에 맞추어서 모든 데이터를 Padding한다.

```
1 max_len=70
2 from keras.preprocessing.sequence import pad_sequences
3 pad_X = pad_sequences(data_X, padding='post', maxlen=max_len)
4 # data_X의 모든 샘플들의 길이를 맞출 때 뒤의 공간에 숫자 0으로 채움.
5 pad_y = pad_sequences(data_y, padding='post', maxlen=max_len)
6 # data_y의 모든 샘플들의 길이를 맞출 때 뒤의 공간에 숫자0으로 채움.
```

Using TensorFlow backend.

```
1 print(min(len(l) for l in pad_X)) # 모든 데이터에서 길이가 가장 짧은 샘플의 길이 출력
2 print(min(len(l) for l in pad_y)) # 모든 데이터에서 길이가 가장 짧은 샘플의 길이 출력
```

70

70

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- **pad\_x**와 **pad\_y**를 훈련 데이터와 테스트 데이터로 8:2로 분할한다.
- **random\_state**를 지정했기 때문에 기존의 **pad\_x**와 **pad\_y**에서 순서가 섞이면서 훈련 Data와 Test Data로 분할된다.

```
1 from sklearn.model_selection import train_test_split  
2 X_train, X_test, y_train, y_test = train_test_split(pad_X, pad_y, test_size=.2, random_state=777)  
  
1 print(len(X_train), len(X_test), len(y_train), len(y_test))
```

11232 2809 11232 2809

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

- 모델 설계를 위한 필요한 도구들을 Import 한다.

```
1 from keras.models import Sequential  
2 from keras.layers import Dense, Embedding, LSTM, Bidirectional, TimeDistributed  
3 from keras.optimizers import Adam
```

```
1 n_words = len(word_to_index)  
2 n_labels = len(ner_to_index)
```

```
1 model = Sequential()  
2 model.add(Embedding(input_dim=n_words, output_dim=16, input_length=max_len, mask_zero=True))  
3 model.add(Bidirectional(LSTM(32, return_sequences=True)))  
4 model.add(TimeDistributed(Dense(n_labels, activation='softmax')))
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 n_words = len(word_to_index)
2 n_labels = len(ner_to_index)
```

```
1 model = Sequential()
2 model.add(Embedding(input_dim=n_words, output_dim=16, input_length=max_len, mask_zero=True))
3 model.add(Bidirectional(LSTM(32, return_sequences=True)))
4 model.add(TimeDistributed(Dense(n_labels, activation='softmax')))
```

- Many-to-Many 문제이므로 **LSTM()**에 **return\_sequences=True**를 설정했다.
- 실습과 같이 각 Data의 길이가 달라서 Padding을 하느라 숫자 0이 많아질 경우에는 **Embedding()**에 **mask\_zero=True**를 설정하여 Data에서 숫자 0은 Padding을 의미하므로 연산에서 제외시킨다는 Option을 줄 수 있다.

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 model.compile(loss='categorical_crossentropy', optimizer=Adam(0.001), metrics=['accuracy'])
```

- 훈련 Data에 대해서 One-hot Encoding을 진행한다.

```
1 from keras.utils import np_utils  
2 y_train2 = np_utils.to_categorical(y_train)  
3 y_train2[0][0]
```

```
array([0., 0., 0., 1., 0., 0., 0., 0., 0.], dtype=float32)
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 model.fit(X_train, y_train2, epochs=8)
```

```
WARNING:tensorflow:From /home/instructor/anaconda3/lib/python3.7/site-packages/keras/backend/tensorflow_backend.py:1033: The name tf.assign_add is deprecated. Please use tf.compat.v1.assign_add instead.
```

Epoch 1/8

```
11232/11232 [=====] - 41s 4ms/step - loss: 0.8660 - acc: 0.8280
```

Epoch 2/8

```
11232/11232 [=====] - 36s 3ms/step - loss: 0.4692 - acc: 0.8557
```

Epoch 3/8

```
11232/11232 [=====] - 35s 3ms/step - loss: 0.3638 - acc: 0.8867
```

Epoch 4/8

```
11232/11232 [=====] - 36s 3ms/step - loss: 0.2997 - acc: 0.9096
```

Epoch 5/8

```
11232/11232 [=====] - 36s 3ms/step - loss: 0.2395 - acc: 0.9298
```

Epoch 6/8

```
11232/11232 [=====] - 36s 3ms/step - loss: 0.1947 - acc: 0.9445
```

Epoch 7/8

```
11232/11232 [=====] - 36s 3ms/step - loss: 0.1656 - acc: 0.9529
```

Epoch 8/8

```
11232/11232 [=====] - 36s 3ms/step - loss: 0.1464 - acc: 0.9579
```

```
<keras.callbacks.History at 0x7f617732fe80>
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 y_test2 = np_utils.to_categorical(y_test)
2 print("테스트 정확도: %.4f" % (model.evaluate(X_test, y_test2)[1]))
```

2809/2809 [=====] - 3s 1ms/step

테스트 정확도: 0.9530

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

```
1 import numpy as np
2
3 index_to_word={}
4 for key, value in word_to_index.items():
5     index_to_word[value] = key
6
7 index_to_ner={}
8 for key, value in ner_to_index.items():
9     index_to_ner[value] = key
10
11
12 i=10 # 확인하고 싶은 테스트용 샘플의 인덱스.
13 y_predicted = model.predict(np.array([X_test[i]])) # 입력한 테스트용 샘플에 대해서 예측 y를 리턴
14 y_predicted = np.argmax(y_predicted, axis=-1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.
15 true = np.argmax(y_test2[i], -1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.
16
17 print("{:15} | {:5} | {}".format("단어", "실제값", "예측값"))
18 print(35 * "-")
19
20 for w, t, pred in zip(X_test[i], true, y_predicted[0]):
21     if w != 0: # PAD값은 제외함.
22         print("{:17}: {:7} {}".format(index_to_word[w], index_to_ner[t], index_to_ner[pred]))
```

# Bi-directional LSTM으로 개체명 인식기 만들기 (Cont.)

단어	실제값	예측값
sarah	: B-PER	B-PER
brady	: I-PER	I-PER
,	: 0	0
whose	: 0	0
republican	: B-MISC	B-MISC
husband	: 0	0
was	: 0	0
OOV	: 0	0
OOV	: 0	0
in	: 0	0
an	: 0	0
OOV	: 0	0
attempt	: 0	0
on	: 0	0
president	: 0	0
ronald	: B-PER	B-PER
reagan	: I-PER	I-PER
,	: 0	0
took	: 0	0
centre	: 0	0
stage	: 0	0
at	: 0	0
the	: 0	0
democratic	: B-MISC	B-MISC
national	: I-MISC	I-MISC
convention	: I-MISC	0

## 4. Part-of-speech Tagging using Bi-LSTM



# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기

- NLTK를 이용하면 English Corpus에 Token化와 품사 Tagging 전처리를 진행한 총 3,914개의 문장 Data를 받아올 수 있다.
- 여기서는 해당 Data를 훈련시켜 품사 Tagging을 수행하는 Model을 만들 어보자.
- 문장 Data를 Download 받아 이 중 첫 번째 문장만 출력해본다.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 import nltk  
2 nltk.download('treebank')
```

```
[nltk_data] Downloading package treebank to  
[nltk_data]      /home/instructor/nltk_data...  
[nltk_data]  Unzipping corpora/treebank.zip.
```

True

```
1 import nltk  
2 tagged_sentences = nltk.corpus.treebank.tagged_sents() # 토큰화에 품사 태깅이 된 데이터 받아오기  
3 print(tagged_sentences[0]) # 첫번째 문장 샘플 출력  
4 print("품사 태깅이 된 문장 개수: ", len(tagged_sentences)) # 문장 샘플의 개수 출력
```

```
[('Pierre', 'NNP'), ('Vinken', 'NNP'), ('.', '.', '.'), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'),  
('.', '.', '.'), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'D'  
T), ('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')  
품사 태깅이 된 문장 개수: 3914
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 훈련을 시키려면 훈련 Data에서 단어에 해당되는 부분과 품사 Tagging 정보에 해당되는 부분을 분리시켜야 한다.
- 즉, `[('Pierre', 'NNP'), ('Vinken', 'NNP')]`와 같은 문장 샘플이 있다면 Pierre과 Vinken을 같이 저장하고, NNP와 NNP를 같이 저장할 필요가 있다.
- 이런 경우 Python 함수 중에서 `zip()` 함수가 유용한 역할을 한다.
- `zip()` 함수는 동일한 개수를 가지는 Sequence 자료형에서 각 순서에 등장하는 원소들끼리 묶어주는 역할을 한다.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 import numpy as np
2
3 sentences, pos_tags = [], []
4 for tagged_sentence in tagged_sentences: # 3,914개의 문장 샘플을 1개씩 불러온다.
5     sentence, tag_info = zip(*tagged_sentence)
6     # 각 샘플에서 단어는 sentence에 품사 태깅 정보는 tags에 저장한다.
7     sentences.append(np.array(sentence)) # 각 샘플에서 단어 정보만 저장한다.
8     pos_tags.append(np.array(tag_info)) # 각 샘플에서 품사 태깅 정보만 저장한다.
```

```
1 print(sentences[0])
2 print(pos_tags[0])
```

```
['Pierre' 'Vinken' ',', '61' 'years' 'old' ',', 'will' 'join' 'the' 'board'
 'as' 'a' 'nonexecutive' 'director' 'Nov.' '29' '.']
['NNP' 'NNP' ',', 'CD' 'NNS' 'JJ' ',', 'MD' 'VB' 'DT' 'NN' 'IN' 'DT' 'JJ'
 'NN' 'NNP' 'CD' '.', ]
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 첫번째 Sample에 대해서 단어에 대해서만 **sentences [0]**에, 또한 품사에 대해서만 **pos\_tags [0]**에 저장된 것을 볼 수 있다.
- sentences**는 예측을 위한 **x**에 해당되며 **pos\_tags**는 예측 대상인 **y**에 해당된다.
- 다른 Sample들에 대해서도 처리가 되었는지 확인하기 위해 임의로 네 번째 Sample에 대해서도 확인해본다.

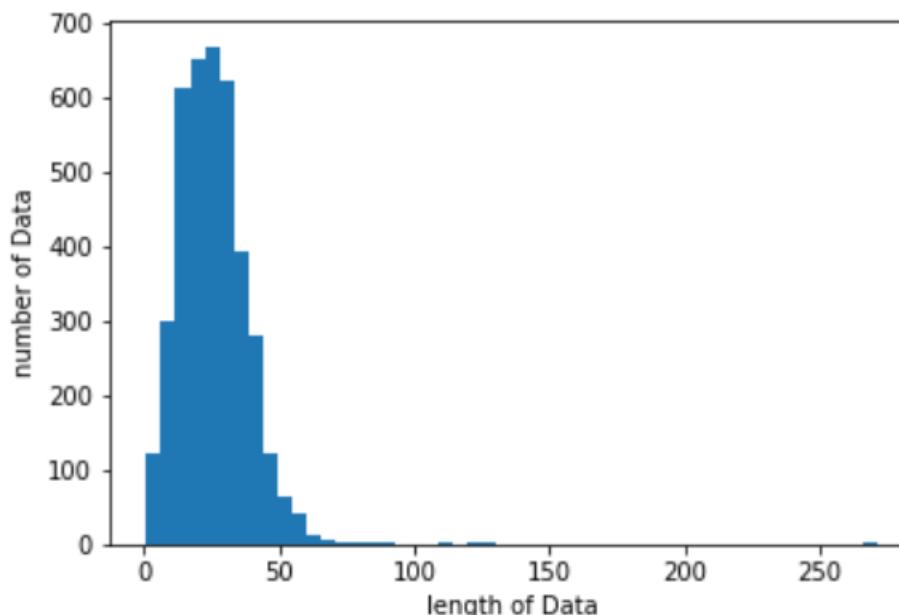
```
1 print(sentences[3])
2 print(pos_tags[3])
```

```
['A' 'form' 'of' 'asbestos' 'once' 'used' '*' '*' 'to' 'make' 'Kent'
 'cigarette' 'filters' 'has' 'caused' 'a' 'high' 'percentage' 'of'
 'cancer' 'deaths' 'among' 'a' 'group' 'of' 'workers' 'exposed' '*' 'to'
 'it' 'more' 'than' '30' 'years' 'ago' ',', 'researchers' 'reported' '0'
 '*T*-1' '.']
['DT' 'NN' 'IN' 'NN' 'RB' 'VBN' '-NONE-' '-NONE-' 'TO' 'VB' 'NNP' 'NN'
 'NNS' 'VBZ' 'VBN' 'DT' 'JJ' 'NN' 'IN' 'NN' 'NNS' 'IN' 'DT' 'NN' 'IN'
 'NNS' 'VBN' '-NONE-' 'TO' 'PRP' 'RBR' 'IN' 'CD' 'NNS' 'IN' ',', 'NNS'
 'VBD' '-NONE-' '-NONE-' '.']
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 문장 Sample의 길이는 전부 다르다.
- 시각화를 통해 확인한다.

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3
4 plt.hist([len(s) for s in sentences], bins=50)
5 plt.xlabel('length of Data')
6 plt.ylabel('number of Data')
7 plt.show()
```



# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 앞의 Graph는 Sample의 길이가 대부분 0~50의 길이를 가지는 것을 알 수 있다.
- 훈련 Data에서 **sentences**를 **x**로, **pos\_tags**를 **y**로 하여 **x**와 **y**를 분리했다.
- 이제 **x**에 대한 단어 집합과 **y**에 대한 단어 집합을 만든다.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 from collections import Counter
2 vocab=Counter()
3 tag_set=set()
4
5 for sentence in sentences: # 훈련 데이터 X에서 문장 샘플을 1개씩 꺼내온다.
6     for word in sentence: # 샘플에서 단어를 1개씩 꺼내온다.
7         vocab[word.lower()]=vocab[word.lower()]+1 # 각 단어의 빈도수를 카운트한다.
8
9 for tags_list in pos_tags: # 훈련 데이터 y에서 품사 태깅 정보 샘플을 1개씩 꺼내온다.
10    for tag in tags_list: # 샘플에서 품사 태깅 정보를 1개씩 꺼내온다.
11        tag_set.add(tag) # 각 품사 태깅 정보에 대해서 중복을 허용하지 않고 집합을 만든다.
```

```
1 print(len(vocab)) # X 데이터의 단어 집합의 길이 출력
2 print(len(tag_set)) # y 데이터의 단어 집합의 길이 출력
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 각 단어에 대해서 Index를 부여하고, 각 품사 Tagging 정보에 대해서도 Index를 부여한다.

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

```
[('(', 4885), ('the', 4764), ('.', 3828), ('of', 2325), ('to', 2182), ('a', 1988), ('in', 1769), ('and', 1556), ('*-1', 1123), ('0', 1099), ('*', 965), ("'s", 865), ('for', 853), ('that', 848), ('*t*-1', 806), ('*u*', 744), ('$', 718), ('``', 702), ('''', 684), ('is', 672), ('said', 628), ('it', 577), ('on', 508), ('%', 446), ('by', 440), ('at', 430), ('as', 415), ('with', 398), ('from', 391), ('million', 383), ('mr.', 375), ('*-2', 372), ('are', 369), ('was', 367), ('be', 356), ('*t*-2', 345), ('its', 343), ('has', 339), ('an', 335), ('new', 328), ('have', 325), ("n't", 325), ('but', 309), ('he', 303), ('or', 294), ('will', 281), ('they', 263), ('company', 260), ('--', 230), ('which', 225), ('this', 224), ('u.s.', 221), ('says', 217), ('year', 214), ('about', 212), ('would', 209), ('more', 204), ('were', 197), ('market', 186), ('their', 184), ('than', 181), ('stock', 172), (';', 171), ('who', 167), ('trading', 167), ('had', 165), ('also', 163), ('president', 161), ('billion', 159), ('up', 153), ('one', 153), ('been', 150), ('some', 146), ('::', 142), ('other', 140), ('not', 140), ('program', 140), ('*-3', 130), ('his', 126), ('because', 124), ('if', 121), ('could', 121), ('share', 120), ('corp.', 117), ('all', 117), ('years', 115), ('i', 115), ('first', 114), ('shares', 114), ('-rrb-', 112), ('two', 109), ('any', 108), ('york', 107), ('-lrb-', 106), ('last', 104), ('there', 100), ('many', 100), ('no', 98), ('such', 98), ('when', 98), ('she', 98), ('inc.', 97), ('*t*-3', 97), ('we', 95), ('can', 95), ('you', 94), ('so', 93), ('japanese', 91), ('after', 89), ('do', 89), ('prices', 89), ('into', 87), ('government', 87), ('&', 86), ('business', 86), ('over', 85), ('most', 83), ('only', 83), ('may', 82), ('sales', 78), ('out', 78), ('these', 77), ('even', 76), ('federal', 76), ('say', 75), ('japan', 75), ('make', 74), ('co.', 74), ('under', 74), ('while', 74), ('board', 73), ('''', 73), ('index', 73), ('recent', 72), ('bia', 72), ('exchange', 72), ('price', 72)]
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 단어의 Index를 부여하기 위한 기준으로는 빈도수를 사용했다.
- 이를 위해 기존에 만든 단어 집합을 빈도수가 큰 순서대로 정렬했다.
- 앞 장의 결과를 보면, ','가 4,885번 등장하여 최다 빈도수를 가진 단어이고, 두 번째로는 **the**가 4,764번 등장하여 두 번째로 빈도수가 높은 단어임을 확인할 수 있다.
- 이제 빈도수가 높은 순서대로 정렬된 단어 집합에 대해서 순차적으로 Index를 부여한다.
- 단, 이 때 뒤에서 모든 문장 Sample의 길이를 맞추기 위한 **PAD**와 모르는 단어를 의미하는 **OOV**도 넣어준다.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 word_to_index={'PAD' : 0, 'OOV' :1}
2 i=1
3 # 인덱스 0은 각각 입력값들의 길이를 맞추기 위한 PAD(padding을 의미)라는 단어에 사용된다.
4 # 인덱스 1은 모르는 단어를 의미하는 OOV라는 단어에 사용된다.
5 for (word, frequency) in vocab_sorted :
6     # if frequency > 1 :
7     # 빈도수가 1인 단어를 제거하는 것도 가능하겠지만 이번에는 별도 수행하지 않고 해보겠음.
8     # 참고로 제거를 수행할 경우 단어 집합의 크기가 절반 정도로 줄어듬.
9     i=i+1
10    word_to_index[word]=i
11 print(word_to_index)
12 print(len(word_to_index))
```

```
{'PAD': 0, 'OOV': 1, ',': 2, 'the': 3, '.': 4, 'of': 5, 'to': 6, 'a': 7, 'in': 8, 'and': 9, '*-1': 10, '0': 11, '*': 12, "'s": 13, 'for': 14, 'that': 15, '*t*-1': 16, '*u*': 17, '$': 18, '''': 19, """": 20, 'is': 21, 'said': 22, 'it': 23, 'on': 24, '%': 25, 'by': 26, 'at': 27, 'as': 28, 'with': 29, 'from': 30, 'million': 31, 'mr.': 32, '*-2': 33, 'are': 34, 'was': 35, 'be': 36, '*t*-2': 37, 'it s': 38, 'has': 39, 'an': 40, 'new': 41, 'have': 42, "n't": 43, 'but': 44, 'he': 45, 'or': 46, 'wil l': 47, 'they': 48, 'company': 49, '--': 50, 'which': 51, 'this': 52, 'u.s.': 53, 'says': 54, 'yea r': 55, 'about': 56, 'would': 57, 'more': 58, 'were': 59, 'market': 60, 'their': 61, 'than': 62, 'st ock': 63, ';': 64, 'who': 65, 'trading': 66, 'had': 67, 'also': 68, 'president': 69, 'billion': 70, 'up': 71, 'one': 72, 'been': 73, 'some': 74, '\'': 75, 'other': 76, 'not': 77, 'program': 78, '*-3': 79, 'his': 80, 'because': 81, 'if': 82, 'could': 83, 'share': 84, 'corp.': 85, 'all': 86, 'years': 87, 'i': 88, 'first': 89, 'shares': 90, '-rrb-': 91, 'two': 92, 'any': 93, 'york': 94, '-lrb-': 95, 'last': 96, 'there': 97, 'many': 98, 'no': 99, 'such': 100, 'when': 101, 'she': 102, 'inc.': 103, '*t*-3': 104, 'we': 105, 'can': 106, 'you': 107, 'so': 108, 'japanese': 109, 'after': 110, 'do': 111,
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 기존 단어 집합의 길이는 11,387였으나 **PAD**와 **oov**를 추가함으로 길이가 11,389가 된 것을 확인할 수 있다.
- 이제 **word\_to\_index**를 통해 단어를 입력하면, Index를 Return 받을 수 있다.

```
1 word_to_index['the']
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- Tagging 정보에 Index를 부여하고, 입력 받은 품사 Tagging 정보에 대해서 Index를 리턴하는 **tag\_to\_index**를 만들어본다.

```
1 tag_to_index={'PAD' : 0}
2 i=0
3 for tag in tag_set:
4     i=i+1
5     tag_to_index[tag]=i
6 print(tag_to_index)
```

```
{'PAD': 0, 'RB': 1, 'EX': 2, 'NNPS': 3, 'VBP': 4, 'RBR': 5, 'NNP': 6, 'IN': 7, 'VBN': 8, 'VBG': 9, 'POS': 10, 'WP$': 11, ' ':' 12, 'VBZ': 13, 'JJR': 14, '#': 15, 'WRB': 16, 'FW': 17, '-NONE-': 18, '-RRB-': 19, '.': 20, 'PRP$': 21, 'MD': 22, 'TO': 23, 'CC': 24, 'WDT': 25, 'NN': 26, 'VBD': 27, 'UH': 28, 'WP': 29, '``': 30, 'RBS': 31, 'CD': 32, ',': 33, 'VB': 34, '$': 35, 'LS': 36, 'PRP': 37, 'DT': 38, 'SYM': 39, 'JJ': 40, 'NNS': 41, '-LRB-': 42, 'JJS': 43, 'PDT': 44, 'RP': 45, "'''": 46}
```

- 총 46개의 단어를 가진 단어 집합에 대해서 Index가 부여됐다.
- Padding을 위해서 **PAD**라는 단어에는 **Index 0**을 부여했다.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 단어 PAD가 추가되면서 단어 집합의 크기는 46에서 47이 되었다.

```
1 len(tag_to_index)
```

47

- **tag\_to\_index**에다가 품사 Tagging 정보를 입력하면 Index를 Return받을 수 있다.

```
1 tag_to_index['UH']
```

28

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- **word\_to\_index**를 사용하여 단어에 대한 훈련 데이터인 **data\_X**를 만든다.

```
1 data_X = []
2
3 for s in sentences:
4     temp_X = []
5     for w in s:
6         try:
7             temp_X.append(word_to_index.get(w.lower(), 1))
8         except KeyError:
9             # 단어 집합을 만들 때 별도로 단어를 제거하지 않았기 때문에
10            # OI 과정에서는 OOV가 존재하지는 않음.
11             temp_X.append(word_to_index['OOV'])
12
13     data_X.append(temp_X)
14 print(data_X)
```

[5602, 3747, 2, 2025, 87, 332, 2, 47, 2406, 3, 132, 28, 7, 2026, 333, 460, 2027, 4], [32, 3747, 21, 178, 5, 5603, 2916, 2, 3, 2917, 638, 148, 4], [2918, 5604, 2, 1137, 87, 332, 9, 603, 178, 5, 3748, 1047, 893, 894, 2, 35, 484, 10, 7, 2026, 333, 5, 52, 1048, 436, 2919, 4], [7, 639, 5, 1049, 640, 324, 12, 12, 6, 128, 1377, 2407, 1550, 39, 895, 7, 191, 1050, 5, 1248, 1760, 233, 7, 148, 5, 515, 3749, 12, 6, 23, 58, 62, 210, 87, 280, 2, 813, 284, 11, 16, 4], [3, 1049, 5605, 2, 2028, 2, 21, 3750, 5606, 640, 23, 2408, 3, 5607, 2, 29, 124, 2029, 2920, 6, 23, 2030, 5608, 15, 16, 814, 71, 2921, 815, 2, 813, 22, 11, 37, 4], [2031, 103, 2, 3, 292, 5, 41, 2032, 5609, 85, 15, 37, 1051, 1377, 1761, 2, 1551, 716, 2028, 8, 38, 5610, 2407, 1550, 8, 3751, 4], [398, 1762, 1378, 59, 284, 33, 58, 62, 7, 55, 280, 2, 3, 516, 517, 2409, 8, 383, 13, 41, 485, 1052, 5, 1249, 2, 7, 3752, 518, 12, 6, 1250, 41, 1379, 6, 3, 353, 4], [7, 2031, 5611, 22, 2, 19, 52, 21, 40, 332, 1552, 4], [105, 368, 2410, 56, 87, 280, 201, 1763, 2411, 5, 1049, 975, 93, 2922, 5612, 4], [97, 21, 99, 1049, 8, 412, 313, 143, 4, 20], [976, 203

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 변화가 되었는지 보기 위해서 첫 번째 Sample에 대해서만 기존의 단어 Sequence를 출력해본다.

```
1 index_to_word={}
2 for key, value in word_to_index.items(): # 인덱스를 단어로 바꾸기 위해 index_to_word를 생성
3     index_to_word[value] = key
4
5
6 temp = []
7 for index in data_X[0]: # 첫번째 문장 샘플 안의 인덱스들에 대해서
8     temp.append(index_to_word[index]) # 다시 단어로 변환
9
10 print(sentences[0]) # 기존 문장 샘플 출력
11 print(temp) # 기존 문장 샘플 → 정수 인코딩 → 복원
```

```
['Pierre', 'Vinken', '61', 'years', 'old', 'will', 'join', 'the', 'board',
 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']
['pierre', 'vinken', '61', 'years', 'old', 'will', 'join', 'the', 'board', 'as', 'a', 'none',
 'xecutive', 'director', 'nov.', '29', '.']
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 기존의 출력 결과와 일치하는 것으로 보아 정상 변환되었음을 알 수 있다.
- 그리고 이제 **y**에 해당하는 전체 Data를 담고 있는 **pos\_tags**에서 품사 Tagging 정보에 해당되는 부분을 모아 **data\_y**에 저장하는 작업을 수행한다.

```
1 data_y = []
2
3 for s in pos_tags:
4     temp_y = []
5     for w in s:
6         temp_y.append(tag_to_index.get(w))
7
8     data_y.append(temp_y)
9
10 print(data_y[0])
```

```
[6, 6, 33, 32, 41, 40, 33, 22, 34, 38, 26, 7, 38, 40, 26, 26, 6, 32, 20]
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 이제 단어에 대한 훈련 데이터인 **data\_x**와 품사 Tagging 정보에 대한 훈련 Data인 **data\_y**가 만들어졌다.
- **data\_x**의 단어 Sequence Data 각각과 **data\_y** 단어 Sequence Sample 각각은 전부 길이가 다르다.
- 양방향 LSTM Model에 손쉽게 Data를 입력으로 사용하기 위해서, 여기서는 모든 Sample의 길이를 동일하게 맞추도록 한다.
- 이에 따라 가장 길이가 긴 Sample의 길이를 우선 구해보자.

```
1 print(max([len(i) for i in data_X])) # 모든 데이터에서 길이가 가장 긴 샘플의 길이 출력
2 print(max([len(i) for i in data_y])) # 모든 데이터에서 길이가 가장 긴 샘플의 길이 출력
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 길이가 가장 긴 샘플의 길이는 271이다.
- 하지만, 앞서 본 Graph에 따르면, 대부분의 Sample은 길이가 50이하이다.
- X에 해당되는 데이터 **data\_x**의 샘플들과 y에 해당되는 데이터 **data\_y** 샘플들의 모든 길이를 임의로 150정도로 맞추려고 한다.
- 이를 위해서 Keras의 **pad\_sequences()**를 사용한다.

```
1 max_len=150
2 from keras.preprocessing.sequence import pad_sequences
3 pad_X = pad_sequences(data_X, padding='post', maxlen=max_len)
4 # data_X의 모든 샘플의 길이를 맞출 때 뒤의 공간에 숫자 0으로 채움.
5 pad_y = pad_sequences(data_y, padding='post', value=tag_to_index['PAD'], maxlen=max_len)
6 # data_y의 모든 샘플의 길이를 맞출 때 뒤의 공간에 'PAD'에 해당되는 인덱스로 채움.
7 # 참고로 숫자 0으로 채우는 것과 'PAD'에 해당하는 인덱스로 채우는 것은 결국 0으로 채워지므로 같음
```

Using TensorFlow backend.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 전체 훈련 Data에서 길이가 가장 짧은 Sample의 길이를 출력했을 때 150이 나오는지 확인해보자.

```
1 print(min([len(i) for i in pad_X])) # 모든 데이터에서 길이가 가장 짧은 샘플의 길이 출력
2 print(min([len(i) for i in pad_y])) # 모든 데이터에서 길이가 가장 짧은 샘플의 길이 출력
```

150  
150

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- `pad_x`와 `pad_y`를 훈련 Data와 Test Data로 8:2로 분할한다.
- `random_state`를 지정했기 때문에 기존의 `pad_x`와 `pad_y`에서 순서가 섞이면서 훈련 Data와 Test Data로 분할된다.
- 즉, 이제 `x_train`의 첫 번째 Sample과 `pad_x`에서의 첫 번째 Sample은 서로 다른 Sample일 수 있다.

```
1 from sklearn.model_selection import train_test_split  
2 X_train, X_test, y_train, y_test = train_test_split(pad_X, pad_y, test_size=.2, random_state=777)
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 훈련 Data `y_train`에 대해서 One-hot Encoding을 수행하고 `y_train2`에 저장한다.

```
1 from keras.utils import np_utils  
2 y_train2 = np_utils.to_categorical(y_train)
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 from keras.models import Sequential
2 from keras.layers import Dense, LSTM, InputLayer, Bidirectional, TimeDistributed, Embedding
3 from keras.optimizers import Adam
4
5 n_words = len(word_to_index)
6 n_labels = len(tag_to_index)
7
8 model = Sequential()
9 model.add(Embedding(n_words, 128, input_length=max_len, mask_zero=True))
10 model.add(Bidirectional(LSTM(256, return_sequences=True)))
11 model.add(TimeDistributed(Dense(n_labels, activation='softmax'))))
12 model.compile(loss='categorical_crossentropy', optimizer=Adam(0.001), metrics=['accuracy'])
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 model.fit(X_train, y_train2, batch_size=128, epochs=6)
```

```
Epoch 1/6
3131/3131 [=====] - 62s 20ms/step - loss: 3.3469 - acc: 0.1519
Epoch 2/6
3131/3131 [=====] - 58s 18ms/step - loss: 2.8595 - acc: 0.2279
Epoch 3/6
3131/3131 [=====] - 60s 19ms/step - loss: 2.3386 - acc: 0.4438
Epoch 4/6
3131/3131 [=====] - 59s 19ms/step - loss: 1.3785 - acc: 0.6542
Epoch 5/6
3131/3131 [=====] - 57s 18ms/step - loss: 0.6854 - acc: 0.8522
Epoch 6/6
3131/3131 [=====] - 55s 18ms/step - loss: 0.3430 - acc: 0.9258
<keras.callbacks.History at 0x7f2d4450c278>
```

```
1 y_test2 = np_utils.to_categorical(y_test)
2 print("테스트 정확도: %.4f" % (model.evaluate(X_test, y_test2)[1]))
```

```
783/783 [=====] - 7s 9ms/step
```

테스트 정확도: 0.9099

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

- 실제로 맞추고 있는지를 특정 Test Data를 주고 직접 출력해서 확인해보자.
- 우선 Index로부터 단어와 품사 Tagging 정보를 Return하는  
`index_to_word`와 `index_to_tag`를 만들고 이를 이용하여 실제값과 예측값을 출력한다.

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

```
1 import numpy as np
2
3 index_to_word={}
4 for key, value in word_to_index.items():
5     index_to_word[value] = key
6
7 index_to_tag={}
8 for key, value in tag_to_index.items():
9     index_to_tag[value] = key
10
11
12 i=10 # 확인하고 싶은 테스트용 샘플의 인덱스.
13 y_predicted = model.predict(np.array([X_test[i]])) # 입력한 테스트용 샘플에 대해서 예측 y를 리턴
14 y_predicted = np.argmax(y_predicted, axis=-1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.
15 true = np.argmax(y_test2[i], -1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.
16
17 print("{:15} | {:5} | {}".format("단어", "실제값", "예측값"))
18 print(35 * "-")
19
20 for w, t, pred in zip(X_test[i], true, y_predicted[0]):
21     if w != 0: # PAD값은 제외함.
22         print("{:17}: {:7} {}".format(index_to_word[w], index_to_tag[t], index_to_tag[pred]))
```

# 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기 (Cont.)

단어	실제값	예측값
in	: IN	IN
addition	: NN	NN
,	: ,	,
buick	: NNP	NNP
is	: VBZ	VBZ
a	: DT	DT
relatively	: RB	RB
respected	: VBN	VBN
nameplate	: NN	NN
among	: IN	IN
american	: NNP	NNP
express	: NNP	NNP
card	: NN	NN
holders	: NNS	NNS
,	: ,	,
says	: VBZ	VBZ
0	: -NONE-	-NONE-
*t*-1	: -NONE-	-NONE-
an	: DT	DT
american	: NNP	NNP
express	: NNP	NNP
spokeswoman	: NN	NN