

# 워드 임베딩

최 석 재

*lingua@naver.com*

# Word Embedding

- Word Embedding은 단어를 숫자로 바꾸는 기법 중 하나이다
  - 특정 단어의 주변의 단어를 이용하여 유사도 즉 숫자로 단어를 대체한다
  - 컴퓨터와 모니터는 한 문장의 주변에 등장하는 비율이 자전거보다 높으므로 더욱 유사하다
- 
- 어제 컴퓨터와 모니터를 구입했다
  - 컴퓨터는 저렴하나, 모니터는 비싸다
  - 나는 자전거 타기를 좋아한다
  - 동생의 자전거 실력도 좋다



id	단어1	단어2	단어3	단어4
1	어제	컴퓨터	모니터	구입하다
2	컴퓨터	저렴하다	모니터	비싸다
3	나	자전거	타다	좋아하다
4	동생	자전거	실력	좋다

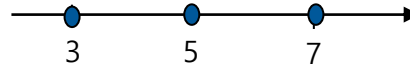
# 워드 임베딩의 기능

- 워드 임베딩은 그동안 풀기 어려웠던 유사어 찾기 문제를 상당히 해결해 주었다
- 유사어는 물론, 반의어를 찾는 문제에서도 어느 정도 성능을 보여준다
- 워드 임베딩을 분류 문제에 적용시키기도 한다
- 워드 임베딩을 이용하면 유사한 단어들을 같은 부류로 처리할 수 있고,
- 원-핫 인코딩에 비하여 저차원 밀집벡터를 사용하게 되어 공간 효율성이 높아진다
- 그러나 워드 임베딩을 통한 유사어 찾기는 저빈도 어휘에서는 낮은 성능을 보이므로 분류 모델에 적용했을 때 반드시 결과가 좋은 것은 아니다
- 여기서는 유사어 찾기와
- 사전 훈련된 임베딩을 이용한 이진 분류
- 학습 데이터로 훈련된 임베딩을 이용한 이진 분류 세 가지를 다루어본다

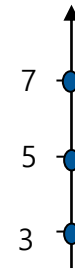
# 차원 이해

- 워드 임베딩은 하나의 단어를 여러 차원을 이용해 표현한다
- 아이폰, 아이패드, 맥북은 하나의 차원을 이용해 다음과 같이 표현할 수 있다
- 아이폰은 아이패드와는 2만큼, 맥북과는 4만큼의 거리에 있고, 아이패드와 맥북은 2만큼의 거리에 있다는 것은 알게 되나, 구조가 단순하여 그 유사도가 정밀히 표현되지 못한다

	x
아이폰	3
아이패드	5
맥북	7



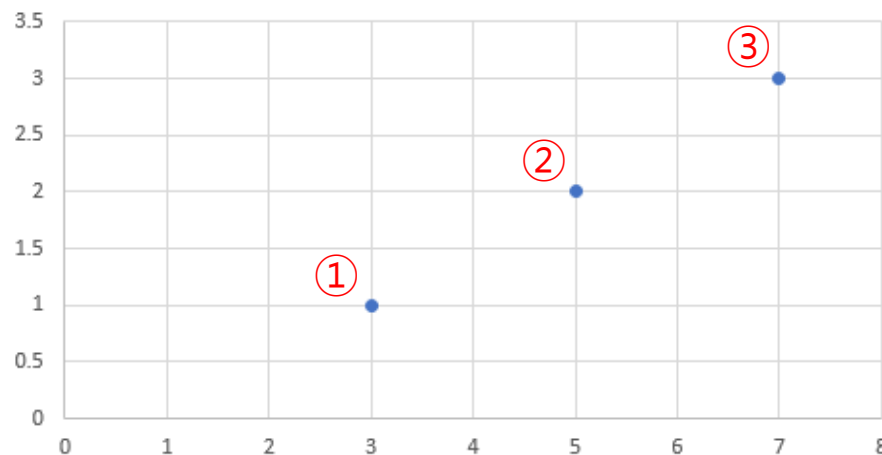
OR



# 차원의 표현

- 데이터에 컬럼을 하나 더 추가하면 데이터의 상호 관계는 2차원이 된다  
①이 ③보다는 ②와 더 가깝다는 것이 더 정밀히 표현된다

	x	y
① 아이폰	3	1
② 아이패드	5	2
③ 맥북	7	3

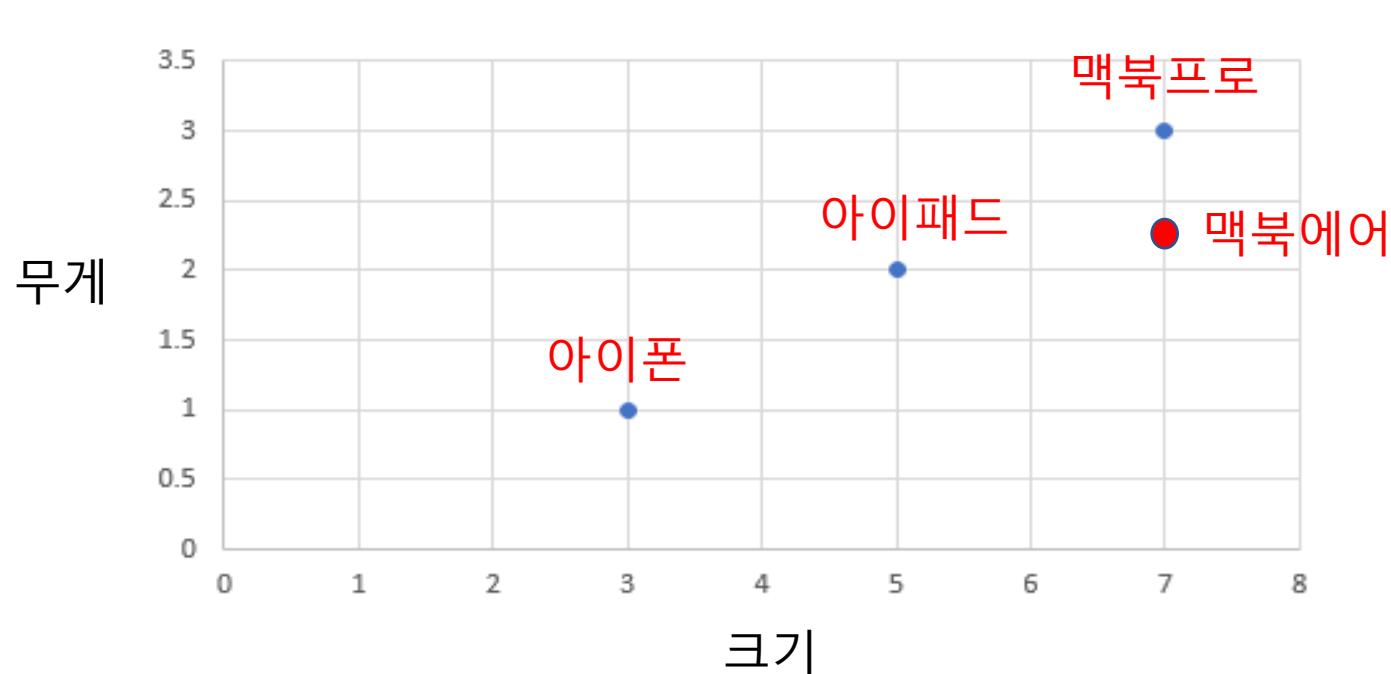


[3, 1]  
[5, 2]  
[7, 3], ... 로 표현된다

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 + \dots}$$

# 차원의 의미

- 유사한 단어는 차원의 값이 유사한 패턴을 가지게 된다
- 이를 차원이 단어의 특정 의미를 표현한다고 해석할 수 있다
- 즉, 단어의 특징이 한 개 혹은 복수의 차원을 통하여 표현된다

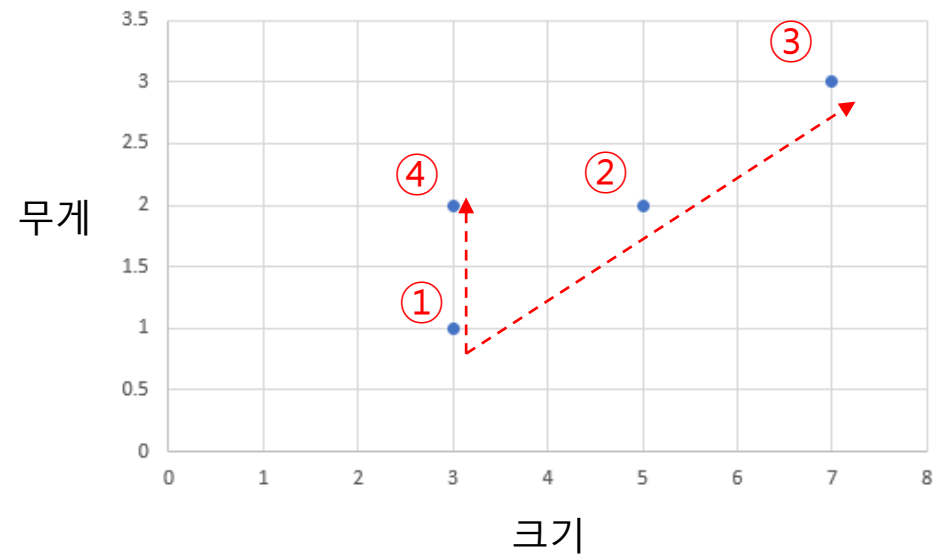


맥북에어는  
맥북프로와 크기는 같으나,  
무게는 아이패드보다 더 가볍다

# 차원의 방향성

- 또한 각 컬럼은 2차원부터는 공간에서 방향성을 가질 수 있다
- 아이폰, 아이패드, 맥북은 크기에 따라 무게가 증가하나,  
어댑터는 크기는 작아도 무게가 많이 나가 다른 패턴을 보인다

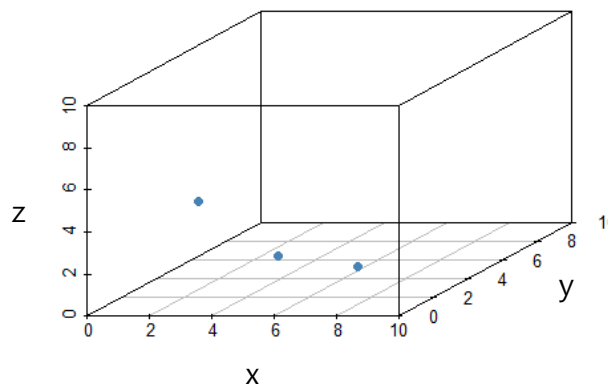
	x	y
① 아이폰	3	1
② 아이패드	5	2
③ 맥북	7	3
④ 어댑터	3	2



# 고차원 데이터

- 데이터에 컬럼을 하나 더 추가하면 데이터의 상호 관계는 3차원이 된다
- 차원의 증가로 상호 관계가 더욱 정밀히 표현된다

	x	y	z	
아이폰		3	1	5
아이패드		5	2	2
맥북		7	3	1



[3, 1, 5]  
[5, 2, 2]  
[7, 3, 1], ... 로 표현된다



# 200차원 임베딩

- 워드 임베딩은 이와 같은 방식으로 대량의 문서를 학습하여 모든 단어의 유사도를 복수의 차원으로 기록한다
- 아래는 200차원으로 표현된 워드 임베딩의 일부이다
- 또한, 이 과정에서 A 컬럼의 내용으로 문자열 1차원 배열이었던 것이 B 컬럼부터 200 개 컬럼으로 구성되는 숫자의 2차원으로 축이 하나 늘게 된다 (행, 열)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	URL	-3.86028	0.583065	-1.41235	2.2038	2.964935	-2.59431	-4.98975	-0.51422	-4.45136	1.181666	-2.59429	4.103975	0.085348	1.466408	-3.71594	2.197032	-4.81038	-1.45395	4.047212	3.93262
2	지미	-2.4833	-0.83459	0.584627	-1.30425	1.797614	-0.7812	0.26234	-0.68078	0.707399	-2.06565	-1.12579	-1.29772	-0.49523	-2.10672	-0.73768	-0.30362	-0.57012	0.972167	-0.21675	1.375072
3	카터	-0.84035	-0.85469	0.518926	-0.10214	2.211183	0.357932	0.458723	-0.2294	-0.84622	-0.33504	-0.91033	-2.11496	2.195074	-1.51444	-0.2925	-1.05111	0.523236	-0.31038	-0.8907	1.779895
4	제임스	1.079353	-0.19112	1.971397	-1.03508	1.296451	-0.58205	-0.58068	-0.3487	0.645975	-0.48476	-0.79937	-0.57293	0.2104	-2.9712	0.154823	-0.70083	-1.70367	1.071476	0.787426	1.670715
5	주니어	0.325491	-1.70622	0.820898	-0.39193	0.548579	-0.5329	-1.80435	-0.21155	1.499797	-0.99638	-1.12531	-0.90812	-0.57182	-0.09192	-0.25568	0.853117	-0.82344	1.625719	-1.01259	-0.47477
6	민주당	2.030566	-1.35959	-1.30362	2.160521	2.770621	-0.05779	-0.78674	-0.377	0.682947	0.147156	2.134828	-1.39753	-0.92722	-1.80073	-2.865	1.695083	0.117571	-1.36661	0.530811	-1.4611
7	출신	0.328542	0.770048	-0.53737	-1.46428	2.693913	-0.52029	-1.66589	-1.09863	-1.23919	0.804363	-0.88631	-1.07253	-3.86398	-0.85849	-2.64378	-0.54486	-1.89279	-1.35291	2.743982	1.066437
8	미국	0.597256	-0.8554	1.6305	-1.12239	2.906073	0.567242	-0.85597	-1.52918	-2.50567	-1.18418	-0.64398	0.499129	1.320328	-0.57343	1.454937	1.11668	-1.75524	0.261288	-0.21519	1.270383
9	대통령	-0.56592	0.974382	1.831127	3.519968	1.585229	2.957404	-0.05195	-2.15176	1.724363	-2.90186	-0.14735	-0.74905	0.208048	0.020867	-0.22852	-1.00402	0.657623	0.754875	0.053104	-0.75876
10	조지아	-0.92856	-0.0968	-2.28289	1.691891	1.408141	-1.35763	1.186497	-0.41216	-1.38797	-0.42866	0.13944	-1.44736	1.239586	-1.01874	0.983563	-2.98323	-1.73745	2.013527	0.221758	1.669652
11	카운티	0.948146	-1.20009	-1.70282	-1.50095	1.469536	-0.50986	-0.1954	-0.37929	-2.69791	-0.06652	0.590661	-0.29452	0.525049	-2.3384	0.507214	-0.67282	-2.11047	0.205022	-0.88123	1.603405
12	마을	-0.08796	-1.61017	-0.34371	0.767736	0.301211	0.124322	1.948312	1.056427	3.592736	0.928602	-1.81859	1.668903	1.180285	1.3479	2.00116	-0.68329	-1.72106	-1.35175	1.870628	-2.26757

# 유사도에 따른 임베딩 값의 차이

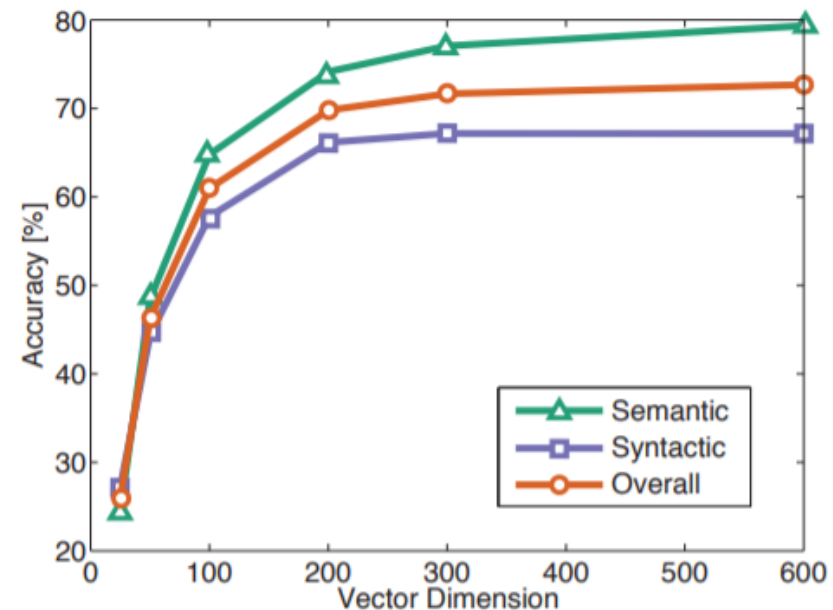
- 컴퓨터와 모니터의 임베딩 값은 자전거와 아래와 같은 차이가 있을 수 있다

	1	2	3	4	5	6	7	8	9	10
컴퓨터	0.1	0.2	0.2	0.1	0.3	0.5	0.4	0.1	0.0	0.0
모니터	0.1	0.2	0.3	0.1	0.3	0.6	0.3	0.1	0.0	0.0
자전거	0.7	0.5	0.0	0.0	0.0	0.2	0.3	0.1	0.5	0.4

- 기계는 '컴퓨터'의 의미를 {0.1, 0.2, 0.2, 0.1, 0.3, ..., 0.0}으로 이해한다
- 임베딩(embedding)이란 자연어(自然語)를 기계가 이해할 수 있는 숫자의 나열인 벡터로 바꾼 결과 또는 벡터로 바꾸는 과정을 의미한다
- 단어나 문장을 벡터로 변환하여 벡터 공간으로 끼워 넣는다(embed)는 의미를 갖는다

# 최적의 Embedding Dimension?

- 한 단어가 몇 개의 컬럼으로 표현되어야 하느냐에 대해서 정답은 없다
- 다만 보통 50~200 차원을 많이 선택하며, 속도가 문제되지 않으면 300 차원을 선택하는 경우도 있다
- 차원을 늘일수록 훈련시간이 늘어나며, 보통 200 차원 이상에서는 훈련 시간은 커지나 성능 개선은 크지 않다



# Word Embedding을 이용한 모델

maxlen = 3, embedding\_dim = 5 인 경우,

# 문장 입력 시 변환 과정

3D (batch\_size, input\_length, feature\_dim)

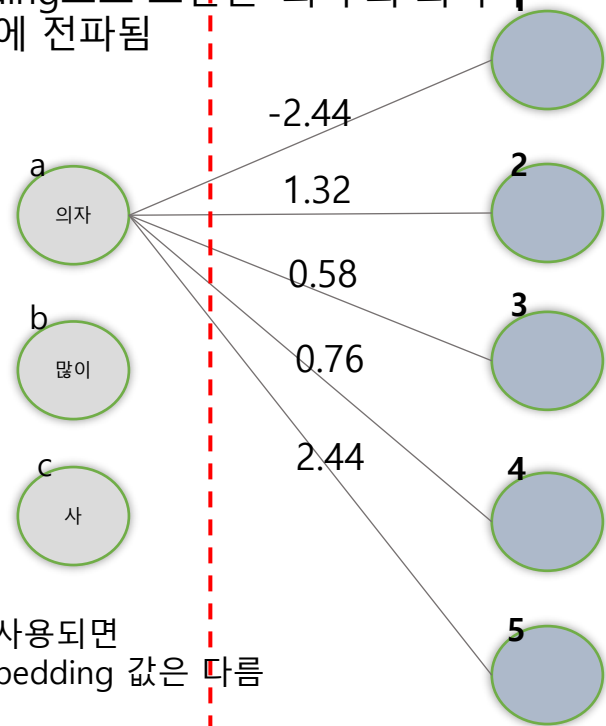
embedding으로 표현된 '의자'의 의미가  
각 노드에 전파됨

Input\_length  
== maxlen  
== 3

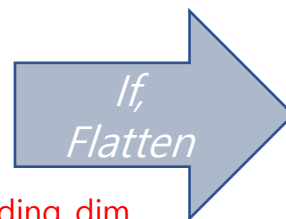
다른 단어가 사용되면  
전파되는 embedding 값은 다름

Padding 되어 3개 단어만 선택되며,  
선택된 3개 단어 각각이 가지고 있는  
5차원의 임베딩 값이 첫 번째 가중치로 배당됨  
이런 식으로 다른 두 단어도 가중치가 배당됨  
이 가중치는 업데이트하지 않고 동결

Embedding 층



Embedding\_dim  
== feature\_dim  
== 5



2D (batch\_size, input\_length x feature\_dim)

a1

a2

a3

...

c5

- 원-핫 인코딩은 max\_words (10,000)을 다 만들고 문장마다 일부만 사용하여 희소벡터가 되지만, 워드임베딩은 모든 노드에 임베딩 값이 들어가므로 밀집벡터라고 한다

- 원-핫 인코딩에서는 사용할 모든 단어의 종류를 늘여놓고, 0과 1을 통하여 그 단어가 사용되었는지를 알려주는 방식인 반면, 임베딩에서는 사용한 단어만 제시하고 그 단어의 의미값을 주는 방식이다

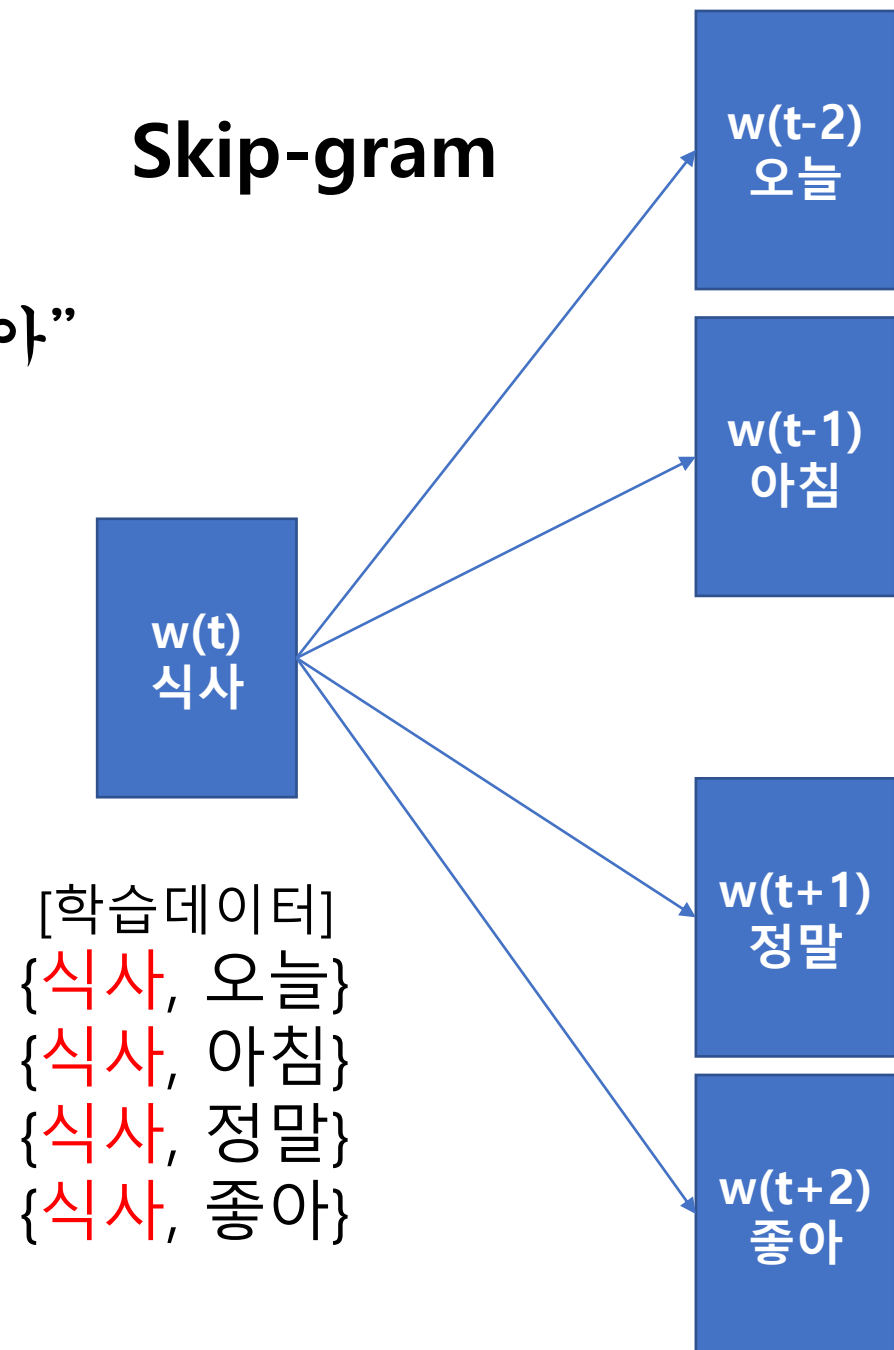
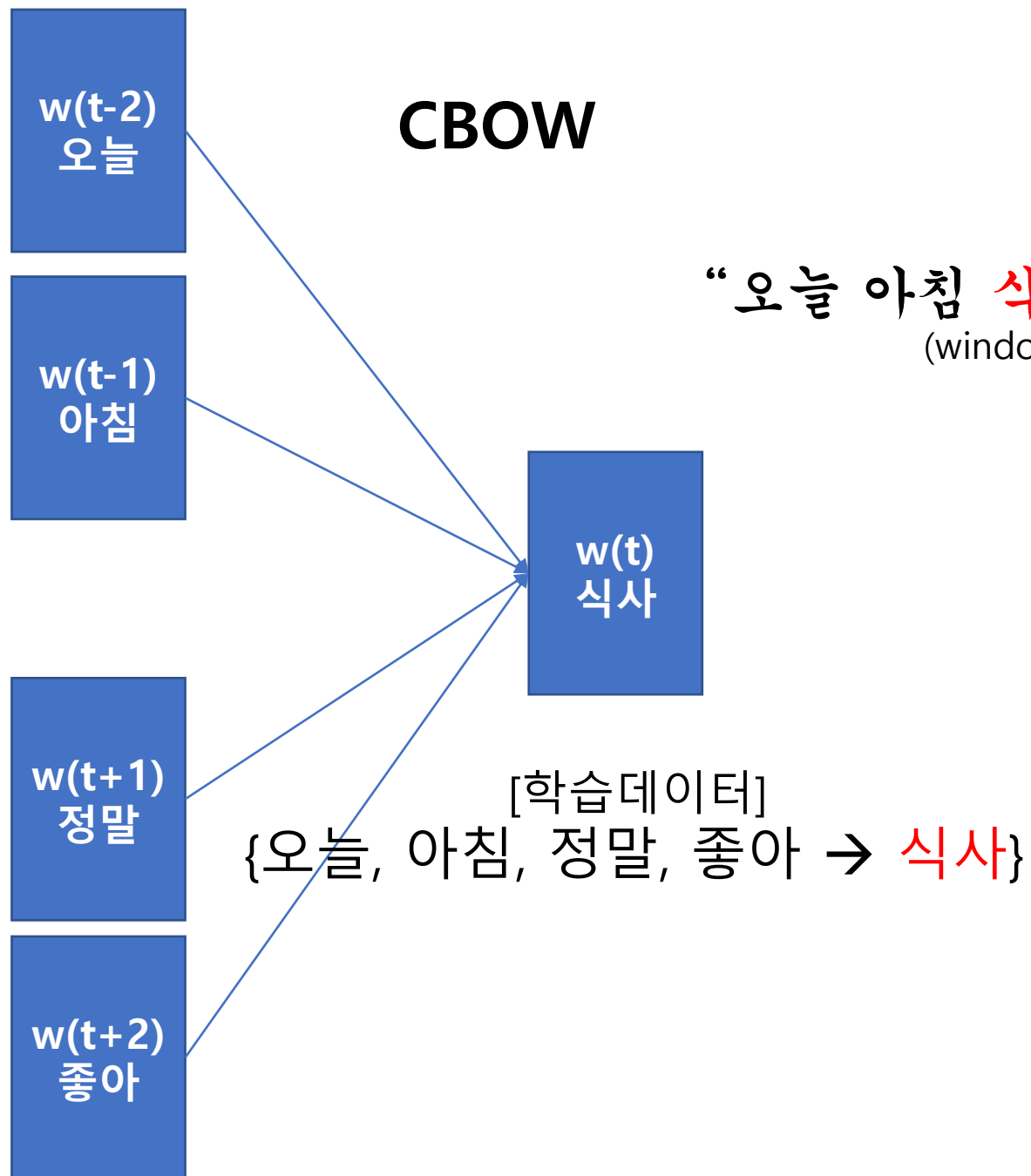
Flatten 하게 되면 두 개의 차원으로 있던 embedding 층이 하나의 층, 15노드로 펼쳐짐

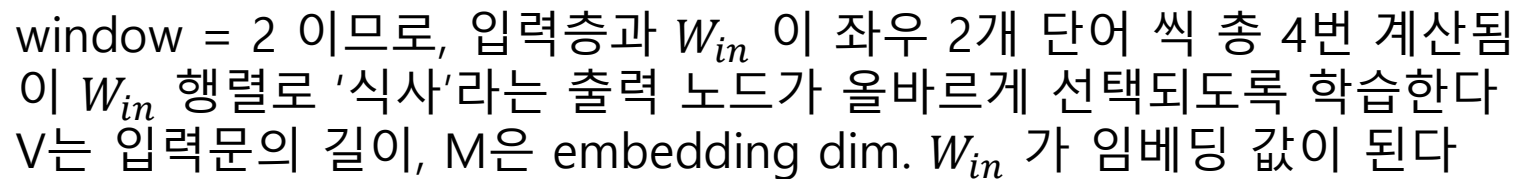
각 단어에 대한 5차원의 임베딩 값이 그 단어의 의미  
즉 임베딩 값으로 현재 문장에서는 어떠한 단어가 들어가고 있는지를 알려준다

예를 들어, '의자 많이 사' 라는 문장은 {-2.44, 1.32, 0.58, 0.76, 2.44, 3.28, -0.51, 1.24, -1.09, 2.11, 0.24, 1.25, -2.44, 1.82, 2.11}의 15개 가중치 노드로 변환된다

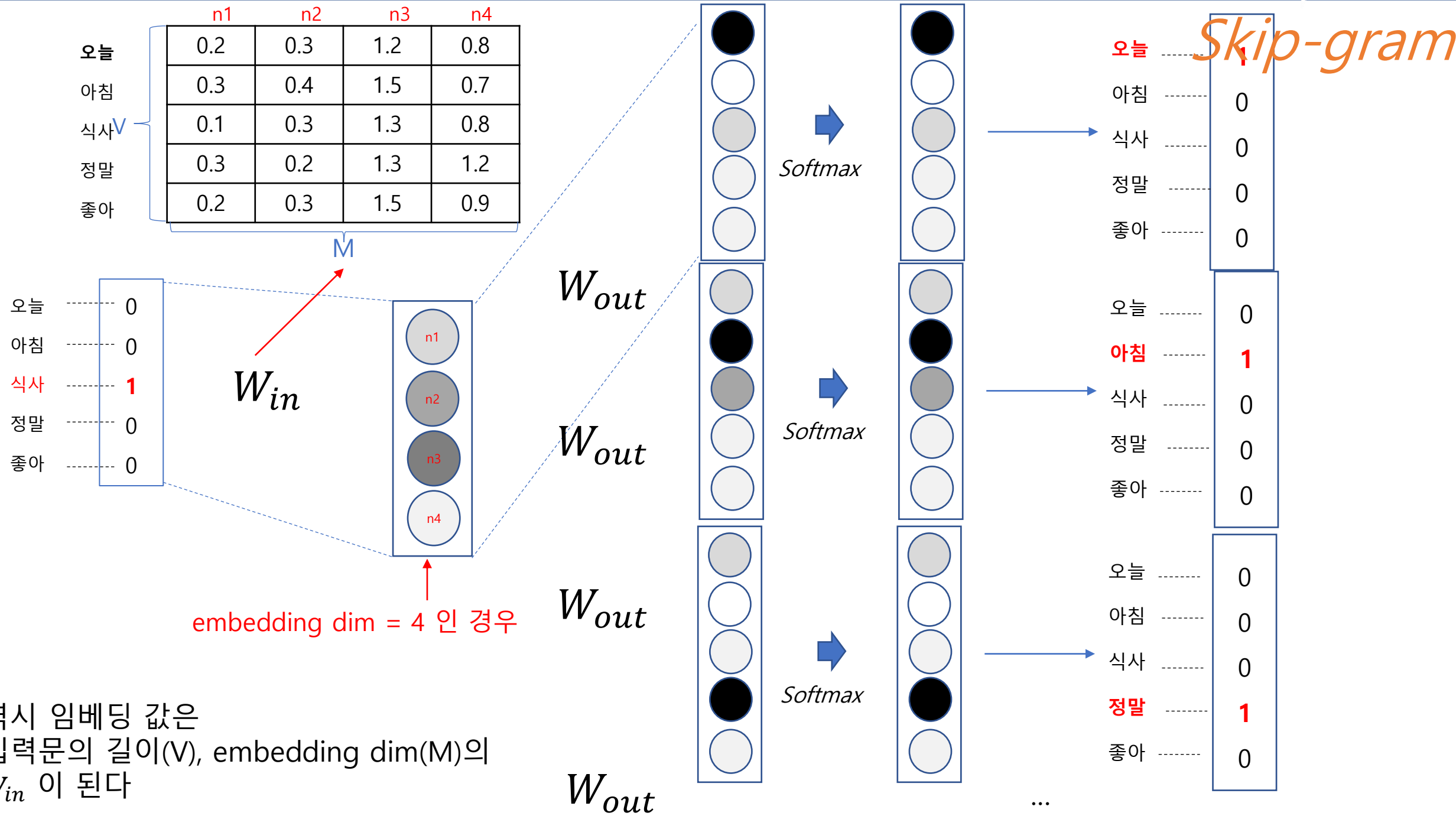
# Word2Vec

- Word2Vec은 구글에서 발표한 가장 기본적인 워드 임베딩 기법
- Word2Vec은 CBOW와 Skip-gram이라는 두 개의 하위 모델이 있다
- CBOW는 주변에 있는 문맥 단어(context word)로 타깃 단어(target word) 하나를 예측하는 방법이고,
- Skip-gram은 타깃 단어를 가지고 주변 문맥 단어를 예측한다
- Skip-gram의 임베딩 품질이 CBOW보다 좋은 경향이 있다









# Word2Vec의 생성과 유사어 찾기

# 구글 드라이브와 연결

```
# from google.colab import auth  
# auth.authenticate_user()
```

- from google.colab import drive
- drive.mount('/content/gdrive')

# 경로 설정

- `pytest_dir = '/content/gdrive/My Drive/pytest/'`
- `chat_dir = '/content/gdrive/My Drive/pytest/data/'`
- `print('pytest_dir:', pytest_dir)`
- `print('chat_dir:', chat_dir)`

```
pytest_dir: /content/gdrive/My Drive/pytest/  
chat_dir: /content/gdrive/My Drive/pytest/data/
```

# 형태소 분석기 관련 설치

- !apt-get update
- !apt-get install g++ openjdk-8-jdk
- !pip install JPytype1
- !pip install rhinoMorph

# 학습할 텍스트 읽기

- `import os.path`
- `embedding_dim = 50` # 임베딩 차원수 설정
- `os.chdir(pytest_dir)` # 경로 설정
- `print("Current Directory:", os.getcwd())`
- `with open('wiki_test.txt', 'r', encoding='utf-8') as f:` # 샘플 파일 읽기  
    `data = f.read()`

Current Directory: /content/gdrive/My Drive/pytest

# 문장단위 분리 및 형태소분석기 기동

- import rhinoMorph
- from nltk.tokenize import sent\_tokenize
- import nltk
- nltk.download('punkt')
- sent\_data = sent\_tokenize(data) # 문장 단위 분리
- rn = rhinoMorph.startRhino() # 형태소분석기 기동
- print('type:', type(sent\_data))
- print('length:', len(sent\_data)) # 전체 문장의 개수
- print('sentence sample:', sent\_data[:20]) # 형태소 분석 전 모습

# 작업 디렉토리 생성

- `import os`
- `exists = os.path.exists(chat_dir+'word2vec')`
- `if not exists:`
  - `os.mkdir(chat_dir+'word2vec')`
  - `print('The word2vec directory is created.')`



# 텍스트의 형태소 분석

- `total_lines = len(sent_data)`
- `cnt = 0`
- `with open(chat_dir+'word2vec/wiki_test_morphed.txt', 'w', encoding='utf-8') as f:`  
    `for data_each in sent_data:`  
        `morphed_data_each = rhinoMorph.onlyMorph_list(rn, data_each, pos=['NNG', 'NNP', 'NP', 'VV', 'VA', 'XR', 'IC', 'MM', 'MAG', 'MAJ'])`  
        `joined_data_each = ' '.join(morphed_data_each)`  
        `if joined_data_each:`  
            `f.write(joined_data_each + '\n')`  
        `cnt += 1`  
        `if (cnt % 1000) == 0:`  
            `print(round(cnt/total_lines * 100, 3), '%')`  
    `print('Morphological Analysis Completed.')`

```
The word2vec directory is created.  
8.349 %  
16.699 %  
25.048 %  
33.397 %  
41.747 %  
50.096 %  
58.445 %  
66.795 %  
75.144 %  
83.493 %  
91.843 %  
Morphological Analysis Completed.
```

# 진행 정도 확인을 위해 1000번째 문장마다 확인

※ 전체 말뭉치 사용을 위해서는 pytest\_자료실.zip 안에 있는 wiki\_data.txt 를 사용

# 형태소 분석 결과를 읽어 리스트로 만들기

- `def read_data(filename, encoding='utf-8'): # 읽기 함수 정의`  
    `with open(filename, 'r', encoding=encoding) as f:`  
        `data = [line.split(' ') for line in f.read().splitlines()]`  
    `return data`
- `data=read_data(chat_dir+"word2vec/wiki_test_morphed.txt", 'utf-8')`
- `print(len(data))`
- `print(type(data))`
- `print(data[:3])`

11976

<class 'list'>

[[ 'URL', '미', '카터', '미', '카터', '제임스', '얼', '미', '카터', '주니어', '민주당', '출신', '미국', '대통령'], ['미', '카터', '조지

# 임베딩 구성

- from gensim.models import Word2Vec
- os.chdir(chat\_dir+'word2vec/')

# size: 차원, window: 컨텍스트 윈도우의 크기, min\_count: 단어 최소 빈도, workers: 학습을 위한 프로세스 수, sg: 0은 CBOW, 1은 skip-gram

- model = Word2Vec(sentences=data, size=embedding\_dim, window=10, min\_count=5, workers=4, sg=1)
- model.save('embedding\_window10\_mincnt5\_skipgram.model')
- print('Completed.')

※ 이 작업의 결과가 매우 큰 경우, 분산 저장을 위하여  
위와 같은 이름의 .model 파일 외에도 .npy 파일이 여러 개 나타난다  
새로 생성된 파일들은 항상 같은 위치에 있도록 한다

# Word2Vec – 임베딩 값 저장

- words = list(model.wv.vocab)
- with open('embedding\_window10\_mincnt5\_skipgram.txt', 'w') as f:

for word in words:

```
data = model.wv[word].tolist()
```

# 현재 단어의 임베딩 값을 가져온다

```
print('data_pre:', data)
```

# 현재 단어의 임베딩 값을 출력해본다

```
data.insert(0, word)
```

# 시작 부분에 해당 단어를 넣는다

```
print('data_after:', data)
```

# 현재 단어의 이름과 함께 임베딩 값을 출력해본다

```
for item in data:
```

# 단어 이름부터 시작하여 각 벡터의 값을 저장한다

```
    f.write("%s " % item)
```

```
f.write("\n")
```

```
data_after: ['민변', -0.18343858420848846, 0.05710616707801819, 0.14069104
data_pre: [0.2339920550584793, 0.03272222727537155, -0.23398496210575104,
data_after: ['안산시', 0.2339920550584793, 0.03272222727537155, -0.2339849
data_pre: [-0.2361794412136078, 0.033876944333314896, 0.15530338883399963,
data_after: ['무효화', -0.2361794412136078, 0.033876944333314896, 0.155303
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.
```

Current values:

NotebookApp.iopub\_data\_rate\_limit=1000000.0 (bytes/sec)

NotebookApp.rate\_limit\_window=3.0 (secs)

# Word2Vec – 유사어 찾기

- `model = Word2Vec.load('embedding_window10_mincnt5_skipgram.model')`
- `print('--- 유사단어 출력 ---')`
- `similarWords = model.wv.most_similar(positive=['행복', '웃음', '밝', '기쁨'], topn=5)`
- `print(similarWords)`
- `word = []`
- `for similarWord in similarWords:` # 유사도값을 제외하고 단어만 모은다  
    `word.append(similarWord[0])`
- `print(word)`

--- 유사단어 출력 ---

```
[('깜짝', 0.9827515482902527), ('숙연', 0.9785732626914978), ('어지럽', 0.9750159382820129), ('살아가', 0.9729760885238647), ('멤돌', 0.9729760885238647)]  
['깜짝', '숙연', '어지럽', '살아가', '멤돌']
```

# Word2Vec – 두 단어 사이의 유사도 계산

- `print('--- 두 단어의 유사도 계산 ---')`
- `print('한국과 일본:', model.wv.similarity('한국', '일본'))`
- `print('한국과 미국:', model.wv.similarity('한국', '미국'))`
- `print('한국과 중국:', model.wv.similarity('한국', '중국'))`

```
--- 두 단어의 유사도 계산 ---  
한국과 일본: 0.64371276  
한국과 미국: 0.7646677  
한국과 중국: 0.81528336
```

# 기 생성된 워드임베딩 사용

※ 여기서부터는 미리 만들어놓은 워드 임베딩(\_big)을 사용한다  
한글위키피디아, 국립국어원 말뭉치, 네이버영화평 파일로 만들었다

- pytest\_big 폴더에 있는 다음의 3개 파일을
- Google Drive의 pytest\data\word2vec 폴더에 업로드한다

 embedding\_window10\_mincnt5\_skipgram\_big.model

 embedding\_window10\_mincnt5\_skipgram\_big

 wiki202003\_nationalcorpus\_naverratings\_morphed\_big

# Word2Vec – 유사어 찾기

- `model = Word2Vec.load('embedding_window10_mincnt5_skipgram_big.model')`
- `print('--- 유사단어 출력 ---')`
- `similarWords = model.wv.most_similar(positive=['행복', '웃음', '밝', '기쁨'], topn=5)`
- `print(similarWords)`
- `word = []`
- `for similarWord in similarWords:` # 유사도값을 제외하고 단어만 모은다  
    `word.append(similarWord[0])`
- `print(word)`

--- 유사단어 출력 ---

```
[('흐뭇', 0.8909333348274231), ('마음', 0.8865163326263428), ('기쁨', 0.8699381351470947), ('마음속', 0.8504815101623535),  
['흐뭇', '마음', '기쁨', '마음속', '즐겁']
```



# Word2Vec – 두 단어 사이의 유사도 계산

- `print('--- 두 단어의 유사도 계산 ---')`
- `print('한국과 일본:', model.wv.similarity('한국', '일본'))`
- `print('한국과 미국:', model.wv.similarity('한국', '미국'))`
- `print('한국과 중국:', model.wv.similarity('한국', '중국'))`

--- 두 단어의 유사도 계산 ---

한국과 일본: 0.85150576

한국과 미국: 0.7342306

한국과 중국: 0.6941229

# 말뭉치에 따른 결과 비교

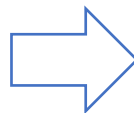
['깜짝', '숙연', '어지럽', '살아가', '멤돌']

--- 두 단어의 유사도 계산 ---

한국과 일본: 0.64371276

한국과 미국: 0.7646677

한국과 중국: 0.81528336



['흐뭇', '마음', '기쁘', '마음속', '즐겁']

--- 두 단어의 유사도 계산 ---

한국과 일본: 0.85150576

한국과 미국: 0.7342306

한국과 중국: 0.6941229

샘플 데이터

국립국어원 말뭉치 + 한글 위키피디아

사전 훈련된 임베딩을 이용한  
이진 분류

# 읽기와 쓰기 함수 정의 및 데이터 로딩

- `os.chdir(pytest_dir)`
- `def read_data(filename, encoding='cp949'):` # 읽기 함수 정의  
    `with open(filename, 'r', encoding=encoding) as f:`  
        `data = [line.split('wt') for line in f.read().splitlines()]`  
        `data = data[1:]` # 첫 행은 헤더(id document label)일 수 있으므로 제외  
    `return data`
- `def write_data(data, filename, encoding='cp949'):` # 쓰기 함수 정의  
    `with open(filename, 'w', encoding=encoding) as f:`  
        `f.write(data)`
- `data = read_data('ratings.txt', encoding='cp949')` # (긍정 10만, 부정 10만)

# 전체 데이터 형태소 분석

```
import rhinoMorph
rn = rhinoMorph.startRhino()

morphed_data = ''
for data_each in data:
    morphed_data_each = rhinoMorph.onlyMorph_list(rn, data_each[1],
        pos=['NNG', 'NNP', 'VV', 'VA', 'XR', 'IC', 'MM', 'MAG', 'MAJ'])
    joined_data_each = ' '.join(morphed_data_each)    # 문자열을 하나로 연결
    if joined_data_each:                               # 내용이 있는 경우만 저장하게 함
        morphed_data += data_each[0] + "Wt" + joined_data_each + "Wt" + data_each[2] + "Wn"

# 형태소 분석된 파일 저장
write_data(morphed_data, 'ratings_morphed.txt', encoding='cp949')
```

# 형태소 분석된 데이터 로딩

```
data = read_data('ratings_morphed.txt' , encoding='cp949')
```

```
print(type(data))
```

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

```
print(len(data[0]))
```

```
print(data[0])
```

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

```
197559
```

```
3
```

```
['8132799', '디자인 배우 학생 외국 디자이너 일구 전통 통하 발전 문화 산업 부럽 사실 우리나라 그 어렵 시절 끝 열정 지키 노라노 같 전통
```

# 데이터 줄이기

# 무료 Colab에서 실행하기에는 데이터가 너무 많아 1/3로 줄인다

- import random
- import math
- import numpy as np
- random.shuffle(data)

# data를 랜덤하게 섞음

- part\_num = math.floor(len(data) \* 1/3)
- data = data[:part\_num]
- print(len(data))

# data의 1/3을 정수로 얻음

# 앞에서부터 1/3 크기의 데이터만 선택

# 65853

# 데이터 분리

# 훈련데이터와 테스트데이터 분리

- data\_text = [line[1] for line in data] # 데이터 본문
- data\_senti = [line[2] for line in data] # 데이터 긍부정 부분
- from sklearn.model\_selection import train\_test\_split
- train\_data\_text, test\_data\_text, train\_data\_senti, test\_data\_senti = train\_test\_split(data\_text, data\_senti, stratify=data\_senti)



# 분리된 데이터 확인

# Counter 클래스를 이용해 train과 test 데이터의 비율을 확인한다

- from collections import Counter
- train\_data\_senti\_freq = Counter(train\_data\_senti)
- print('train\_data\_senti\_freq:', train\_data\_senti\_freq)
- test\_data\_senti\_freq = Counter(test\_data\_senti)
- print('test\_data\_senti\_freq:', test\_data\_senti\_freq)

```
train_data_senti_freq: Counter({'0': 24721, '1': 24668})  
test_data_senti_freq: Counter({'0': 8241, '1': 8223})
```

# Data Tokenizing

# 단어에 숫자 기호를 배당하는 Tokenizing

- from keras.preprocessing.text import Tokenizer
- from keras.preprocessing.sequence import pad\_sequences
- import numpy as np
- import math
  
- max\_words = 10000                   # 데이터셋에서 가장 빈도 높은 10,000 개의 단어만 사용
- maxlen = 20                         # 20개 이후의 단어는 버려 각 문장의 길이를 고정
  
- tokenizer = Tokenizer(num\_words=max\_words) # 상위빈도 10,000 개 단어를 추려내는 Tokenizer 객체 생성
- tokenizer.fit\_on\_texts(train\_data\_text)                   # 단어 인덱스를 구축한다
- word\_index = tokenizer.word\_index                         # 단어 인덱스만 가져온다

# Tokenizer 결과 확인

# Tokenizing 결과 확인

- print('전체에서 %s개의 고유한 토큰을 찾았습니다.' % len(word\_index))
- print('word\_index type: ', type(word\_index))
- print('word\_index: ', word\_index)

전체에서 21947개의 고유한 토큰을 찾았습니다.

word\_index type: <class 'dict'>

word\_index: {'영화': 1, '하': 2, '보': 3, '없': 4, 'ㅋㅋ': 5, '재미있': 6, '좋': 7, '너무': 8, '되': 9, '정말': 10, '있': 11,

# Data Sequencing

# 텍스트를 숫자로 변환

# 상위 빈도 max\_words 개의 단어만 추출하여 word\_index의 숫자 리스트로 변환

- data = tokenizer.texts\_to\_sequences(train\_data\_text) # 데이터에 Tokenizer 적용
- print('data 0:', data[0])
- print('texts 0:', train\_data\_text[0])

```
data 0: [668, 7, 1]
texts 0: 훈훈 좋 영화 ^^
```

# 토큰으로 선정된 각 단어에 대하여 index가 배당된다



# Data Type 확인

- `print(type(train_data_text))`
- `print(type(data))`
- `print(data.shape)`

```
<class 'list'>  
<class 'numpy.ndarray'>  
(49389, 20)
```

# One-Hot Encoding (불필요)

- `def to_one_hot(sequences, dimension):`
- `results = np.zeros((len(sequences), dimension))`
- `for i, sequence in enumerate(sequences):`
- `results[i, sequence] = 1.`
- `return results`

# 워드 임베딩을 사용할 때는 본문에 원-핫 인코딩을 하지 않는다

# 현재는 이진분류이므로 Label에도 원-핫 인코딩을 할 필요가 없다

# `data = to_one_hot(data, dimension=max_words)`

- `labels = np.asarray(train_data_senti).astype('float32')`

# Data 확인

- `print(type(train_data_text))`
- `print(type(data))`
- `print(data.shape)`
- `print('데이터 텐서의 차원:', data.ndim)`
- `print('레이블 텐서의 차원:', labels.ndim)`
- `print('데이터 텐서의 크기:', data.shape)`
- `print('레이블 텐서의 크기:', labels.shape)`

```
<class 'list'>  
<class 'numpy.ndarray'>  
(49389, 20)  
데이터 텐서의 차원: 2  
레이블 텐서의 차원: 1  
데이터 텐서의 크기: (49389, 20)  
레이블 텐서의 크기: (49389,)
```



# 검증 데이터 분리

- `validation_ratio = 0.3` # 30%는 검증데이터로 사용한다. 나머지는 훈련데이터
- `validation_len = math.floor(len(train_data_text) * validation_ratio)`
- `x_train = data[validation_len:]` # 훈련데이터의 70%는 훈련데이터
- `y_train = labels[validation_len:]` # 훈련데이터의 70%는 훈련데이터 Label
- `x_val = data[:validation_len]` # 훈련데이터의 30%는 검증데이터
- `y_val = labels[:validation_len]` # 훈련데이터의 30%는 검증데이터 Label

# 임베딩 딕셔너리 로딩 – Word2Vec Embedding (1)

# 단어와 임베딩 값의 딕셔너리 로딩

- `embeddings_index = {}`
- `f = open(os.path.join(chat_dir+'word2vec', 'embedding_window10_mincnt5_skipgram_big.txt'), encoding='cp949')`
- `for line in f:`
  - `values = line.split()` # 텍스트 파일의 각 행을 분리
  - `word = values[0]` # 각 행의 단어
  - `coefs = np.asarray(values[1:], dtype='float32')` # 각 단어의 임베딩값
  - `embeddings_index[word] = coefs`
- `f.close()`
- `print('%s개의 단어 벡터를 찾았습니다.' % len(embeddings_index))` # 143775

※ 미리 배포된 \_big은 기본값인 CP949(ANSI) 코드로 저장되었다

# 임베딩 행렬 구성 – Word2Vec Embedding (2)

# 임베딩 행렬 구성

```
embedding_matrix = np.zeros((max_words, embedding_dim)) # 0으로 채워진 빈 행렬 구성
for word, i in word_index.items(): # word_index의 word와 index 추출
    if i < max_words: # max_words 이하의 범위에서 순회
        embedding_vector = embeddings_index.get(word) # 해당 단어의 임베딩 벡터 추출
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector # 임베딩 값을 행렬의 해당 word_index 위치에 주입
```

# 파라미터 설정

- `class_number = 1`
- `epochs = 5`
- `batch_size = 32`
- `embedding_dim = 50`
- `model_name = 'text_binary_model.h5'`
- `tokenizer_name = 'text_binary_tokenizer.pickle'`

# 임베딩 층 쌓기 – Word2Vec Embedding (3)

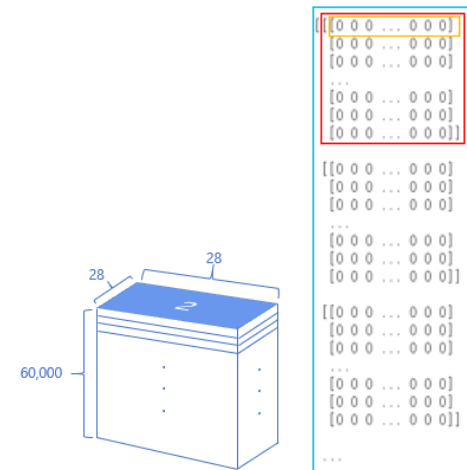
- from keras import models
- from keras import layers
- model = models.Sequential()

# 모델을 새로 정의

- # 임베딩을 하게 되면 embedding\_dim을 값으로 갖는 층이 하나 늘게 된다(2D -> 3D)
- # 따라서 다음에 Dense와 같이 2D를 입력으로 받는 층을 사용하려면 Flatten()을 통해 다시 2D로 차원을 축소한다
- # 만약 다음에 3D(batch\_size, input\_length, feature\_dim)를 입력으로 받는 순환신경망을 이용할 때는 Flatten() 없이 바로 사용하면 된다
- # input\_length(=maxlen)은 나중에 임베딩된 입력을 Flatten 층에서 펼치기 위해 필요한 것이며, Flatten 하지 않는다면 생략 가능하다
- # Flatten 하게 되면 3D 임베딩 텐서를 (batch\_size, input\_length \* output\_dim) 크기의 2D 텐서로 펼쳐 Dense 층의 입력이 될 수 있게 된다

- **model.add(layers.Embedding(input\_dim=max\_words, output\_dim=embedding\_dim, input\_length=maxlen))**
- **model.add(layers.Flatten())**

- model.add(layers.Dense(units=64, activation='relu')) # 은닉층
- model.add(layers.Dense(units=32, activation='relu')) # 은닉층
- model.add(layers.Dense(units=class\_number, activation='sigmoid')) # 출력층



# 모델 요약 출력

- model.summary()

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 20, 50)	500000
flatten (Flatten)	(None, 1000)	0
dense (Dense)	(None, 64)	64064
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 1)	33

=====  
Total params: 566,177  
Trainable params: 566,177  
Non-trainable params: 0  
=====

Layer 1  
(embedding, 20, 50 node)

Layer 2  
(Flatten, 1000 node)

Layer 3  
(dense 0, 64 node)

Layer 4  
(dense 1, 32 node)

Layer 5  
(dense 2, 1 node)

# 임베딩 행렬의 값 주입 – Word2Vec Embedding (4)

- # 임베딩 층에 사전 훈련된 임베딩값 주입
- # 그리고 이 층은 훈련되지 못하도록 동결한다
  - `model.layers[0].set_weights([embedding_matrix])`
  - `model.layers[0].trainable = False`

# Compile Model

# 신경망의 출력이 확률이므로 오차값 계산은 crossentropy를 사용하는 것이 최선이다

# 가중치 업데이트는 RMSprop을 사용하였다.

# crossentropy는 원본의 확률 분포와 예측의 확률 분포를 측정하여 조절해 간다

# 또한 다중 분류이므로 binary\_crossentropy를 사용한다

- `model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])`



# Train Model

- # 32개씩 미니 배치를 만들어 10번의 epoch로 훈련
- # 훈련데이터로 훈련하고, 검증데이터로 검증한다
- # 반환값의 history는 훈련하는 동안 발생한 모든 정보를 담고 있는 딕셔너리이다

- `history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(x_val, y_val), verbose=1)`
- `history_dict = history.history`

```
Epoch 1/5
1081/1081 [=====] - 8s 4ms/step - loss: 0.5465 - acc: 0.7206 - val_loss: 0.5101 - val_acc: 0.7450
Epoch 2/5
1081/1081 [=====] - 4s 3ms/step - loss: 0.4983 - acc: 0.7575 - val_loss: 0.5087 - val_acc: 0.7478
Epoch 3/5
1081/1081 [=====] - 4s 3ms/step - loss: 0.4728 - acc: 0.7731 - val_loss: 0.4953 - val_acc: 0.7563
Epoch 4/5
1081/1081 [=====] - 4s 3ms/step - loss: 0.4447 - acc: 0.7901 - val_loss: 0.5101 - val_acc: 0.7537
Epoch 5/5
1081/1081 [=====] - 4s 3ms/step - loss: 0.4177 - acc: 0.8061 - val_loss: 0.5133 - val_acc: 0.7561
```

# Save Model

# 만들어진 모델을 이후에 재사용할 수 있도록 저장한다

- import pickle
- model.save(model\_name)

# 훈련데이터에서 사용된 상위빈도 max\_words 개의 단어로 된 Tokenizer도 저장 (같은 단어 추출)

- with open(tokenizer\_name, 'wb') as file:
- pickle.dump(tokenizer, file, protocol=pickle.HIGHEST\_PROTOCOL)

# Accuracy & Loss 확인

# history 딕셔너리 안에 있는 정확도와 손실값을 가져와 본다

- `acc = history.history['acc']`
- `val_acc = history.history['val_acc']`
- `loss = history.history['loss']`
- `val_loss = history.history['val_loss']`
- `print('Train accuracy of each epoch:', np.round(acc, 3))`
- `print('Validation accuracy of each epoch:', np.round(val_acc, 3))`
- `epochs = range(1, len(val_acc) + 1)`

```
Train accuracy of each epoch: [0.721 0.757 0.773 0.79  0.806]
```

```
Validation accuracy of each epoch: [0.745 0.748 0.756 0.754 0.756]
```

# Plotting Accuracy

# 정확도와 손실값의 변화를 보고, epoch를 어디에서 조절해야 할 지를 가늠한다.

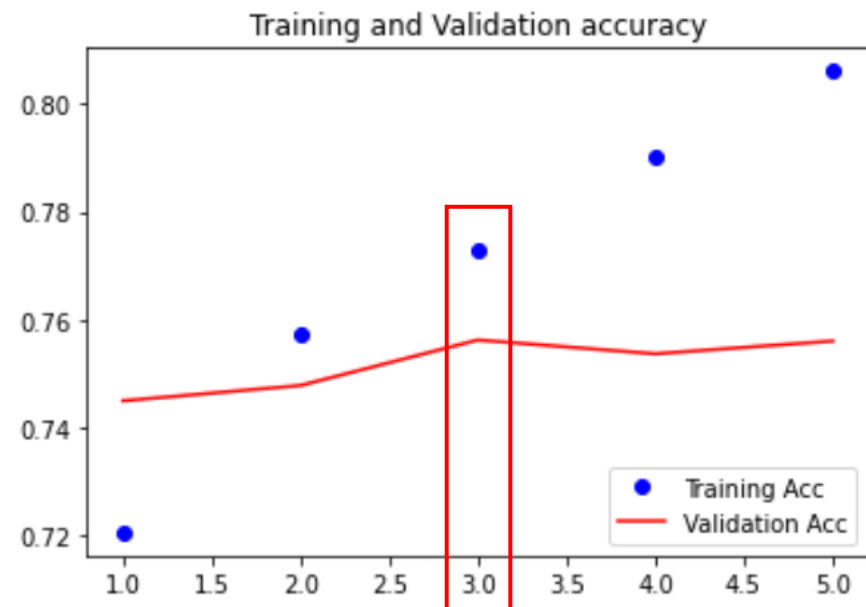
# 정확도가 떨어지는 구간, 손실값이 높게 나타나는 구간을 확인한다

# 데이터가 큰 경우 대개 epoch를 늘려야 최적값에 도달한다

- `import matplotlib.pyplot as plt`

# 정확도 그리기

- `plt.plot(epochs, acc, 'bo', label='Training Acc')`
- `plt.plot(epochs, val_acc, 'b', label='Validation Acc')`
- `plt.title('Training and validation accuracy')`
- `plt.xlabel('Epochs')`
- `plt.ylabel('Accuracy')`
- `plt.legend()`
- `plt.show()`

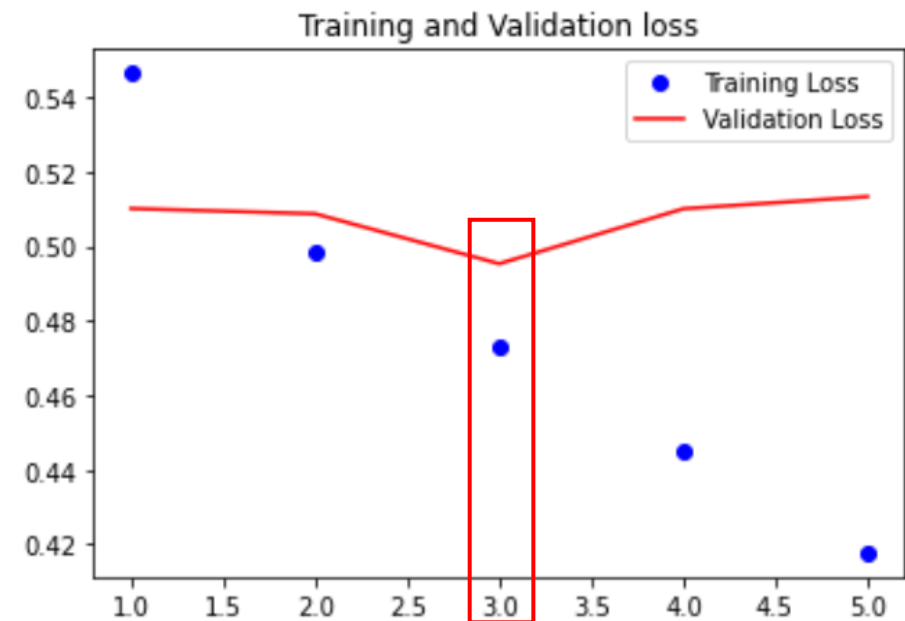


# Plotting Loss

- `plt.figure()` # 새로운 그림을 그린다

# 손실값 그리기

- `plt.plot(epochs, loss, 'bo', label='Training Loss')`
- `plt.plot(epochs, val_loss, 'b', label='Validation Loss')`
- `plt.title('Training and validation loss')`
- `plt.xlabel('Epochs')`
- `plt.ylabel('loss')`
- `plt.legend()`
  
- `plt.show()`



# Load Model

※ 로컬 컴퓨터에서 현재 파일의 폴더를 가져오기  
# 현재 파일이 있는 폴더를 가져온다. 폴더에 한글이 있으면 안됨  
filepath = os.path.dirname(os.path.realpath(\_\_file\_\_))

- from keras.models import load\_model
- loaded\_model = load\_model(model\_name)
- with open(tokenizer\_name, 'rb') as handle:
- loaded\_tokenizer = pickle.load(handle)

# 테스트 데이터 Sequencing

# 문자열을 word\_index의 숫자 리스트로 변환

- data = loaded\_tokenizer.texts\_to\_sequences(test\_data\_text)

# padding으로 문자열의 길이를 고정시킨다

- data = pad\_sequences(data, maxlen=maxlen)

# 원-핫 인코딩은 하지 않는다

# x\_test = to\_one\_hot(data, dimension=max\_words)

x\_test = data

# test\_data\_senti를 list에서 넘파이 배열로 변환

- y\_test = np.asarray(test\_data\_senti).astype('float32')

# 테스트 데이터 평가

# 모델에 분류할 데이터와 그 정답을 같이 넣어준다

- `test_eval = loaded_model.evaluate(x_test, y_test)`

# 모델이 분류한 결과와 입력된 정답을 비교한 결과

- `print('prediction model loss & acc:', test_eval)`

```
515/515 [=====] - 1s 2ms/step - loss: 0.5196 - acc: 0.7546  
prediction model loss & acc: [0.5195907354354858, 0.7546161413192749]
```



# 1개 데이터 예측

# 형태소분석을 포함하여 이제까지의 과정을 모두 진행해주어야 한다

- `text = ["재미있게 잘 봤습니다"]` # 데이터를 list 타입으로 만든다

- `import rhinoMorph`

- `rn = rhinoMorph.startRhino()`

# 리스트 컴프리헨션으로 실질형태소만을 리스트로 가져온다

- `text=[rhinoMorph.onlyMorph_list(m, sentence, pos=['NNG', 'NNP', 'NP', 'VV', 'VA', 'XR', 'IC', 'MM', 'MAG', 'MAJ'], eomi=False) for sentence in text]`

- `print('형태소 분석 결과:', text)`

- `data = loaded_tokenizer.texts_to_sequences(text)`

- `data = pad_sequences(data, maxlen=maxlen)`

- `# x_test = to_one_hot(data, dimension=max_words)`

- `x_test = data`

- `prediction = loaded_model.predict(x_test)`

- `print("Result:", prediction)`

filepath: /usr/local/lib/python3.7/dist-packages

classpath: /usr/local/lib/python3.7/dist-packages/rhinoMorph/lib/rhino.jar

JVM is already started~

RHINO started!

형태소 분석 결과: [['재미있', '잘', '보']]

Result: [[0.95765877]]

1(긍정)일 확률 출력

# 학습 데이터 임베딩을 이용한 이진 분류

※ 기본적으로 앞의 것과 모두 같고,  
다만 Word2Vec Embedding (1), (2), (4)가 없다

# 읽기와 쓰기 함수 정의

- `os.chdir(pytest_dir)`
- `def read_data(filename, encoding='cp949'):` # 읽기 함수 정의
  - `with open(filename, 'r', encoding=encoding) as f:`
  - `data = [line.split('\\\\t') for line in f.read().splitlines()]`
  - `data = data[1:]` # 첫 행은 헤더(id document label)일 수 있으므로 제외
  - `return data`
- `def write_data(data, filename, encoding='cp949'):` # 쓰기 함수 정의
  - `with open(filename, 'w', encoding=encoding) as f:`
  - `f.write(data)`

# 형태소 분석된 데이터 로딩

```
data = read_data('ratings_morphed.txt' , encoding='cp949')
```

```
print(type(data))
```

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

```
print(len(data[0]))
```

```
print(data[0])
```

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

```
197559
```

```
3
```

```
['8132799', '디자인 배우 학생 외국 디자이너 일구 전통 통하 발전 문화 산업 부럽 사실 우리나라 그 어렵 시절 끝 열정 지키 노라노 같 전통
```

# 데이터 줄이기

- import random
- import math
- import numpy as np
- random.shuffle(data)
- part\_num = math.floor(len(data) \* 1/2)
- data = data[:part\_num]
- print(len(data))

# data를 랜덤하게 섞음

# data의 1/2을 정수로 얻음

# 앞에서부터 절반 크기의 데이터만 선택

# 98779

※ 여기서부터는 데이터의 절반을 이용해본다  
메모리 문제가 발생하면 1/3로 줄인다

# 데이터 분리

# 훈련데이터와 테스트데이터 분리

- data\_text = [line[1] for line in data]
- data\_senti = [line[2] for line in data]

# 데이터 본문

# 데이터 긍부정 부분

- from sklearn.model\_selection import train\_test\_split
- train\_data\_text, test\_data\_text, train\_data\_senti, test\_data\_senti = train\_test\_split(data\_text, data\_senti, stratify=data\_senti)

# 분리된 데이터 확인

# Counter 클래스를 이용해 train과 test 데이터의 비율을 확인한다

- from collections import Counter
- train\_data\_senti\_freq = Counter(train\_data\_senti)
- print('train\_data\_senti\_freq:', train\_data\_senti\_freq)
- test\_data\_senti\_freq = Counter(test\_data\_senti)
- print('test\_data\_senti\_freq:', test\_data\_senti\_freq)

```
train_data_senti_freq: Counter({'1': 37068, '0': 37016})  
test_data_senti_freq: Counter({'1': 12356, '0': 12339})
```

# Data Tokenizing

# 단어에 숫자 기호를 배당하는 Tokenizing

- from keras.preprocessing.text import Tokenizer
- from keras.preprocessing.sequence import pad\_sequences
- import numpy as np
- import math
  
- max\_words = 10000                      # 데이터셋에서 가장 빈도 높은 10,000 개의 단어만 사용
- maxlen = 20                              # 20개 이후의 단어는 버려 각 문장의 길이를 고정
  
- tokenizer = Tokenizer(num\_words=max\_words) # 상위빈도 10,000 개 단어를 추려내는 Tokenizer 객체 생성
- tokenizer.fit\_on\_texts(train\_data\_text)                      # 단어 인덱스를 구축한다
- word\_index = tokenizer.word\_index                      # 단어 인덱스만 가져온다



# Tokenizer 결과 확인

# Tokenizing 결과 확인

- `print('전체에서 %s개의 고유한 토큰을 찾았습니다.' % len(word_index))`
- `print('word_index type: ', type(word_index))`
- `print('word_index: ', word_index)`

전체에서 25771개의 고유한 토큰을 찾았습니다.

`word_index type: <class 'dict'>`

`word_index: {'영화': 1, '하': 2, '보': 3, '없': 4, 'ㅋㅋ': 5, '재미있': 6, '너무': 7, '좋': 8, '되': 9, '있': 10, '정말': 11,`

# Data Sequencing

# 텍스트를 숫자로 변환

# 상위 빈도 10,000(max\_words)개의 단어만 추출하여 word\_index의 숫자 리스트로 변환

- data = tokenizer.texts\_to\_sequences(train\_data\_text) # 데이터에 Tokenizer 적용
- print('data 0:', data[0])
- print('texts 0:', train\_data\_text[0])

data 0: [564, 29, 38, 89, 1698, 69, 431, 20]

texts 0: 헐 주 아깍 한국 영화계 이렇게 망하 ㅋ

# 토큰으로 선정된 각 단어에 대하여 index가 배당된다

# Data Padding

- `data = pad_sequences(data, maxlen=maxlen)`
- `print('data:', data)`
- `print('data 0:', data[0])`
- `print('data 0의 길이:', len(data[0]))`

```
data: [[ 0 0 0 ... 69 431 20]
 [ 0 0 0 ... 0 0 7]
 [ 0 0 0 ... 41 7019 33]
 ...
 [ 0 0 0 ... 193 443 233]
 [ 334 3 199 ... 936 2991 1657]
 [ 0 0 0 ... 75 29 3190]]
data 0: [ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 564 29
      38 89 1698 69 431 20]
data 0의 길이: 20
```

# Data Type 확인

- `print(type(train_data_text))`
- `print(type(data))`
- `print(data.shape)`

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

```
<class 'numpy.ndarray'>
```

```
(74084, 20)
```

# One-Hot Encoding (불필요)

- `def to_one_hot(sequences, dimension):`
- `results = np.zeros((len(sequences), dimension))`
- `for i, sequence in enumerate(sequences):`
- `results[i, sequence] = 1.`
- `return results`

# 워드 임베딩을 사용할 때는 본문에 원-핫 인코딩을 하지 않는다

# 현재는 이진분류이므로 Label에도 원-핫 인코딩을 할 필요가 없다

# `data = to_one_hot(data, dimension=max_words)`

- `labels = np.asarray(train_data_senti).astype('float32')`

# Data 확인

- `print(type(train_data_text))`
- `print(type(data))`
- `print(data.shape)`
- `print('데이터 텐서의 차원:', data.ndim)`
- `print('레이블 텐서의 차원:', labels.ndim)`
- `print('데이터 텐서의 크기:', data.shape)`
- `print('레이블 텐서의 크기:', labels.shape)`

```
<class 'list'>
<class 'numpy.ndarray'>
(74084, 20)
데이터 텐서의 차원: 2
레이블 텐서의 차원: 1
데이터 텐서의 크기: (74084, 20)
레이블 텐서의 크기: (74084,)
```

# 검증 데이터 분리

- `validation_ratio = 0.3` # 30%는 검증데이터로 사용한다. 나머지는 훈련데이터
- `validation_len = math.floor(len(train_data_text) * validation_ratio)`
- `x_train = data[validation_len:]` # 훈련데이터의 70%는 훈련데이터
- `y_train = labels[validation_len:]` # 훈련데이터의 70%는 훈련데이터 Label
- `x_val = data[:validation_len]` # 훈련데이터의 30%는 검증데이터
- `y_val = labels[:validation_len]` # 훈련데이터의 30%는 검증데이터 Label

# 파라미터 설정

- `class_number = 1`
- `epochs = 5`
- `batch_size = 32`
- `embedding_dim = 50`
- `model_name = 'text_binary_model.h5'`
- `tokenizer_name = 'text_binary_tokenizer.pickle'`



# 모델 정의하기 – Embedding

- `from tensorflow.keras import models`
- `from tensorflow.keras import layers`

- `embedding_dim = 50`                      # 임베딩의 차원을 설정한다. 보통 50~200까지에서 적절히 설정한다
- `model = models.Sequential()`              # 모델을 새로 정의

# 임베딩을 하게 되면 `embedding_dim`을 값으로 갖는 층이 하나 늘게 된다(2D -> 3D)

# 따라서 다음에 Dense와 같이 2D를 입력으로 받는 층을 사용하려면 `Flatten()`을 통해 다시 2D로 차원을 축소한다

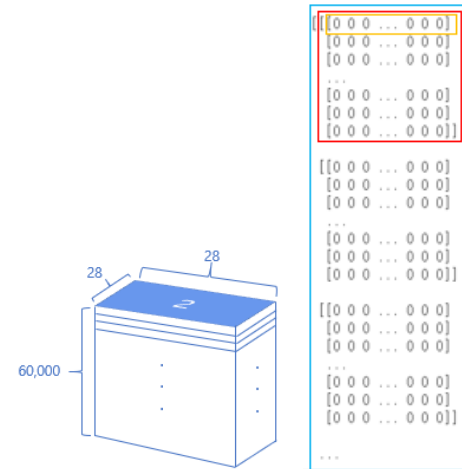
# 만약 다음에 3D(`batch_size`, `input_length`, `feature_dim`)를 입력으로 받는 순환신경망을 이용할 때는 `Flatten()` 없이 바로 사용하면 된다

# `input_length(=maxlen)`은 나중에 임베딩된 입력을 Flatten 층에서 펼치기 위해 필요한 것이며, Flatten 하지 않는다면 생략 가능하다

# Flatten 하게 되면 3D 임베딩 텐서를 (`batch_size`, `input_length * output_dim`) 크기의 2D 텐서로 펼쳐 Dense 층의 입력이 될 수 있게 된다

- **`model.add(layers.Embedding(input_dim=max_words, output_dim=embedding_dim, input_length=maxlen))`**
- **`model.add(layers.Flatten())`**

- `model.add(layers.Dense(units=32, activation='relu'))`                      # 은닉층
- `model.add(layers.Dense(units=class_number, activation='sigmoid'))`              # 출력층



# 모델 요약 출력

- model.summary()

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 20, 50)	500000
flatten (Flatten)	(None, 1000)	0
dense (Dense)	(None, 64)	64064
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 1)	33

Total params: 566,177  
Trainable params: 566,177  
Non-trainable params: 0

Layer 1  
(embedding, 200, 50 node)

Layer 2  
(Flatten, 10000 node)

Layer 3  
(dense 0, 64 node)

Layer 4  
(dense 1, 32 node)

Layer 5  
(dense 2, 1 node)

# Compile Model

# 신경망의 출력이 확률이므로 오차값 계산은 crossentropy를 사용하는 것이 최선이다

# 가중치 업데이트는 RMSprop을 사용하였다.

# crossentropy는 원본의 확률 분포와 예측의 확률 분포를 측정하여 조절해 간다

# 또한 다중 분류이므로 binary\_crossentropy를 사용한다

- `model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])`

# Train Model

- # 32개씩 미니 배치를 만들어 n번의 epoch로 훈련
- # 훈련데이터로 훈련하고, 검증데이터로 검증한다
- # 반환값의 history는 훈련하는 동안 발생한 모든 정보를 담고 있는 딕셔너리이다

- `history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(x_val, y_val), verbose=1)`
- `history_dict = history.history`

```
Epoch 1/5
1621/1621 [=====] - 8s 4ms/step - loss: 0.4391 - acc: 0.7902 - val_loss: 0.3883 - val_acc: 0.8277
Epoch 2/5
1621/1621 [=====] - 7s 4ms/step - loss: 0.3640 - acc: 0.8401 - val_loss: 0.3892 - val_acc: 0.8277
Epoch 3/5
1621/1621 [=====] - 7s 4ms/step - loss: 0.3220 - acc: 0.8635 - val_loss: 0.4069 - val_acc: 0.8229
Epoch 4/5
1621/1621 [=====] - 7s 4ms/step - loss: 0.2749 - acc: 0.8876 - val_loss: 0.4597 - val_acc: 0.8081
Epoch 5/5
1621/1621 [=====] - 7s 4ms/step - loss: 0.2380 - acc: 0.9055 - val_loss: 0.4803 - val_acc: 0.8096
```

# Save Model

# 만들어진 모델을 이후에 재사용할 수 있도록 저장한다

- import pickle
- model.save(model\_name)

# 훈련데이터에서 사용된 상위빈도 1,000개의 단어로 된 Tokenizer 저장(같은 단어를 추출하게 한다)

- with open(tokenizer\_name, 'wb') as file:
- pickle.dump(tokenizer, file, protocol=pickle.HIGHEST\_PROTOCOL)

# Accuracy & Loss 확인

# history 딕셔너리 안에 있는 정확도와 손실값을 가져와 본다

- `acc = history.history['acc']`
- `val_acc = history.history['val_acc']`
- `loss = history.history['loss']`
- `val_loss = history.history['val_loss']`
- `print('Train accuracy of each epoch:', np.round(acc, 3))`
- `print('Validation accuracy of each epoch:', np.round(val_acc, 3))`
- `epochs = range(1, len(val_acc) + 1)`

```
Train accuracy of each epoch: [0.79  0.84  0.863 0.888 0.905]
```

```
Validation accuracy of each epoch: [0.828 0.828 0.823 0.808 0.81 ]
```

# Plotting Accuracy

# 정확도와 손실값의 변화를 보고, epoch를 어디에서 조절해야 할 지를 가늠한다.

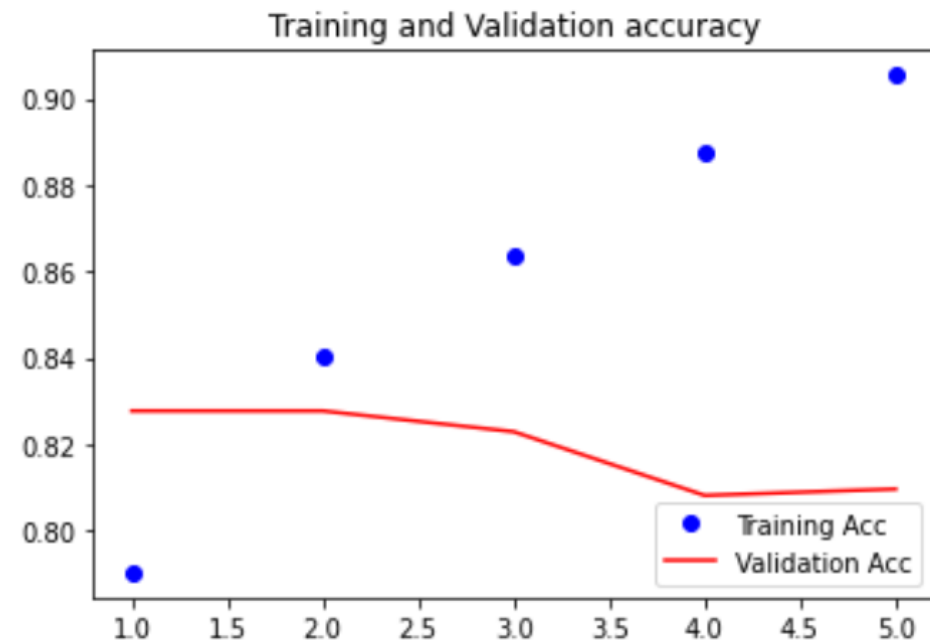
# 정확도가 떨어지는 구간, 손실값이 높게 나타나는 구간을 확인한다

# 데이터가 큰 경우 대개 epoch를 늘려야 최적값에 도달한다

- `import matplotlib.pyplot as plt`

# 정확도 그리기

- `plt.plot(epochs, acc, 'bo', label='Training Acc')`
- `plt.plot(epochs, val_acc, 'b', label='Validation Acc')`
- `plt.title('Training and validation accuracy')`
- `plt.xlabel('Epochs')`
- `plt.ylabel('Accuracy')`
- `plt.legend()`
- `plt.show()`

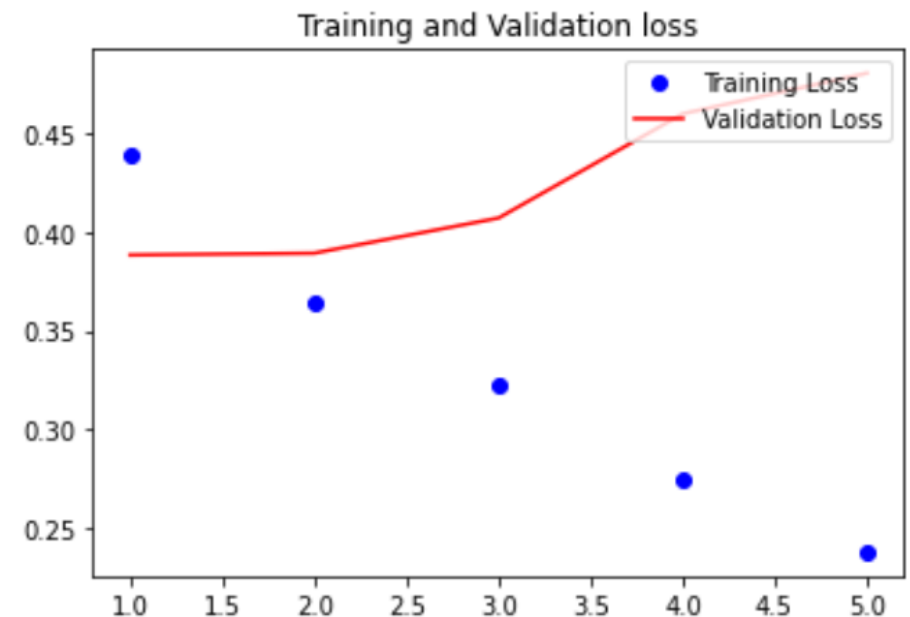


# Plotting Loss

- `plt.figure()` # 새로운 그림을 그린다

# 손실값 그리기

- `plt.plot(epochs, loss, 'bo', label='Training Loss')`
- `plt.plot(epochs, val_loss, 'b', label='Validation Loss')`
- `plt.title('Training and validation loss')`
- `plt.xlabel('Epochs')`
- `plt.ylabel('loss')`
- `plt.legend()`
  
- `plt.show()`





# Load Model

※ 로컬 컴퓨터에서 현재 파일의 폴더를 가져오기  
# 현재 파일이 있는 폴더를 가져온다. 폴더에 한글이 있으면 안됨  
filepath = os.path.dirname(os.path.realpath(\_\_file\_\_))

- from keras.models import load\_model
- loaded\_model = load\_model(model\_name)
- with open(tokenizer\_name, 'rb') as handle:
- loaded\_tokenizer = pickle.load(handle)

# 테스트 데이터 Sequencing

# 문자열을 word\_index의 숫자 리스트로 변환

- data = loaded\_tokenizer.texts\_to\_sequences(test\_data\_text)

# padding으로 문자열의 길이를 고정시킨다

- data = pad\_sequences(data, maxlen=maxlen)

# 원-핫 인코딩은 하지 않는다

# x\_test = to\_one\_hot(data, dimension=max\_words)

x\_test = data

# test\_data\_senti를 list에서 넘파이 배열로 변환

- y\_test = np.asarray(test\_data\_senti).astype('float32')

# 테스트 데이터 평가

# 모델에 분류할 데이터와 그 정답을 같이 넣어준다

- `test_eval = loaded_model.evaluate(x_test, y_test)`

# 모델이 분류한 결과와 입력된 정답을 비교한 결과

- `print('prediction model loss & acc:', test_eval)`

```
772/772 [=====] - 3s 3ms/step - loss: 0.4850 - acc: 0.8094  
prediction model loss & acc: [0.4849987030029297, 0.8093541264533997]
```

# 1개 데이터 예측

# 형태소분석을 포함하여 이제까지의 과정을 모두 진행해주어야 한다

- `text = ["재미있게 잘 봤습니다"]`    # 데이터를 list 타입으로 만든다

- `import rhinoMorph`

- `rn = rhinoMorph.startRhino()`

# 리스트 컴프리헨션으로 실질형태소만을 리스트로 가져온다

- `text=[rhinoMorph.onlyMorph_list(m, sentence, pos=['NNG', 'NNP', 'NP', 'VV', 'VA', 'XR', 'IC', 'MM', 'MAG', 'MAJ'], eomi=False) for sentence in text]`

- `print('형태소 분석 결과:', text)`

- `data = loaded_tokenizer.texts_to_sequences(text)`

- `data = pad_sequences(data, maxlen=maxlen)`

- `# x_test = to_one_hot(data, dimension=max_words)`

- `x_test = data`

- `prediction = loaded_model.predict(x_test)`

- `print("Result:", prediction)`

filepath: /usr/local/lib/python3.7/dist-packages

classpath: /usr/local/lib/python3.7/dist-packages/rhinoMorph/lib/rhino.jar

JVM is already started~

RHINO started!

형태소 분석 결과: [['재미있', '잘', '보']]

Result: [[0.9743759]]

1(긍정)일 확률 출력

# 검증 데이터 성능 비교

※ 모두 데이터를 1/2로 줄여 새로 모델링한 결과

## Embedding 없는 모델

Train accuracy of each epoch: [0.806 0.839 0.848 0.854 0.864]

Validation accuracy of each epoch: [0.823 0.829 0.831 0.828 0.832]

## 사전학습된 임베딩 모델

Train accuracy of each epoch: [0.734 0.764 0.78 0.794 0.81 ]

Validation accuracy of each epoch: [0.746 0.756 0.751 0.762 0.761]

## 학습데이터 임베딩 모델

Train accuracy of each epoch: [0.791 0.841 0.865 0.89 0.906]

Validation accuracy of each epoch: [0.822 0.823 0.819 0.806 0.807]

※ 학습데이터에 대해 임베딩 모델을 200 차원으로 늘려도 [0.824 0.822 0.817 0.808 0.786]로 크게 개선되지 않았음

# 테스트 데이터 성능 비교

## Embedding 없는 모델

772/772 [=====] - 3s 3ms/step - loss: 0.3921 - acc: 0.8277  
prediction model loss & acc: [0.3921140730381012, 0.8277384042739868]

## 사전학습된 임베딩 모델

772/772 [=====] - 2s 2ms/step - loss: 0.5125 - acc: 0.7604  
prediction model loss & acc: [0.5124735236167908, 0.7604373097419739]

## 학습데이터 임베딩 모델

772/772 [=====] - 2s 2ms/step - loss: 0.4866 - acc: 0.8086  
prediction model loss & acc: [0.48659422993659973, 0.8085847496986389]

※ 학습데이터에 대해 임베딩 모델을 200 차원으로 늘려도 [0.5635424852371216, 0.787608802318573]로 개선되지 않았음

# 부록

말뭉치 수집 방법

# 한국어 위키피디아

- 한국어 위키피디아는 500MB 정도의 한국어 자료를 보유한다
- 이는 국립국어원에서 구축한 한국어 형태분석 말뭉치의 크기와 비슷하다
- 다음의 사이트에서 한국어 위키피디아 덤프 파일을 받는다
- <https://dumps.wikimedia.org/kowiki/latest/>

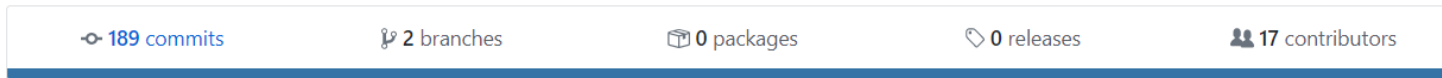
<a href="#">kowiki-latest-pages-articles-multistream5.xml-p...</a>	10-Mar-2020 05:36	868
<a href="#">kowiki-latest-pages-articles-multistream6.xml-p...</a>	03-Mar-2020 08:43	159872671
<a href="#">kowiki-latest-pages-articles-multistream6.xml-p...</a>	10-Mar-2020 05:36	871
<a href="#">kowiki-latest-pages-articles.xml.bz2</a>	02-Mar-2020 22:43	663870976
<a href="#">kowiki-latest-pages-articles.xml.bz2-rss.xml</a>	09-Mar-2020 12:47	791
<a href="#">kowiki-latest-pages-articles1.xml-p1p76864.bz2</a>	02-Mar-2020 21:49	60452380
<a href="#">kowiki-latest-pages-articles1.xml-p1p76864.bz2-...</a>	09-Mar-2020 12:47	811
<a href="#">kowiki-latest-pages-articles2.xml-p76865p239412...</a>	02-Mar-2020 21:52	89389424
<a href="#">kowiki-latest-pages-articles2.xml-p76865p239412...</a>	09-Mar-2020 12:47	826
<a href="#">kowiki-latest-pages-articles3.xml-p239413p51243...</a>	02-Mar-2020 21:56	101347581
<a href="#">kowiki-latest-pages-articles3.xml-p239413p51243...</a>	09-Mar-2020 12:47	829
<a href="#">kowiki-latest-pages-articles4.xml-p512440p91564...</a>	02-Mar-2020 22:01	123787883
<a href="#">kowiki-latest-pages-articles4.xml-p512440p91564...</a>	09-Mar-2020 12:47	829
<a href="#">kowiki-latest-pages-articles5.xml-p915642p16267...</a>	02-Mar-2020 22:07	145872627
<a href="#">kowiki-latest-pages-articles5.xml-p915642p16267...</a>	09-Mar-2020 12:47	832
<a href="#">kowiki-latest-pages-articles6.xml-p1626758p2652...</a>	02-Mar-2020 22:14	142434373
<a href="#">kowiki-latest-pages-articles6.xml-p1626758p2652...</a>	09-Mar-2020 12:47	835

- 많은 종류가 있지만 항상 다음만 필요하다
- kowiki-latest-pages-articles.xml.bz2
- 그 외의 파일은 편집 역사, 편집 내역, 권한 기록 등 실제 내용과는 다른 부분이 포함된 파일이다



- 압축 파일은 xml 형식으로 되어 있기 때문에 일반 텍스트 파일로 변환한다
- 아래의 위키피디아 extractor를 이용하여 간단히 변환할 수 있다
- <https://github.com/attardi/wikiextractor>







A tool for extracting plain text from Wikipedia dumps



189 commits   2 branches   0 packages   0 releases   17 contributors

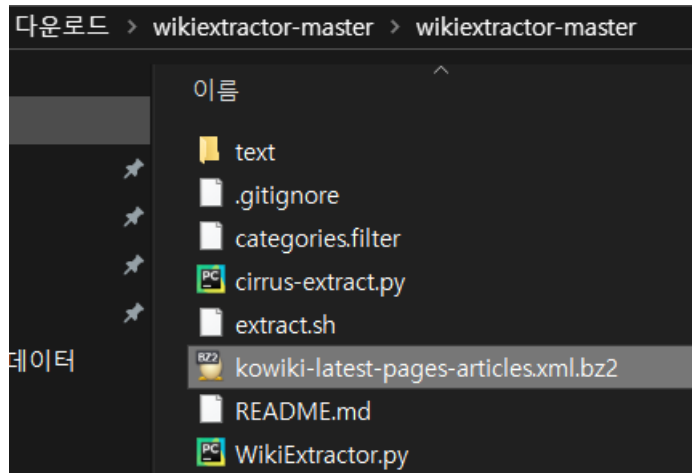
Branch: master   New pull request   Find file   **Clone or download**

attardi Update WikiExtractor.py   Latest commit 16186e2 on 2 Mar

 .gitignore	Merge branch 'add_extra_fields_to_cirrus_output' of https://github.co...	12 months ago
 README.md	log save to file; log page statistic info;	3 years ago
 WikiExtractor.py	Update WikiExtractor.py	last month
 categories.filter	filter_categories use depth 4 under Health	3 years ago
 cirrus-extract.py	extract language and revion from cirrus search	12 months ago
 extract.sh	minimized complexity	2 years ago

download

- Extractor의 압축을 푼 뒤, 위키피디아 파일과 한 폴더에 둔다
- 여기서는 extractor 폴더에 위키피디아 xml 파일을 두었다

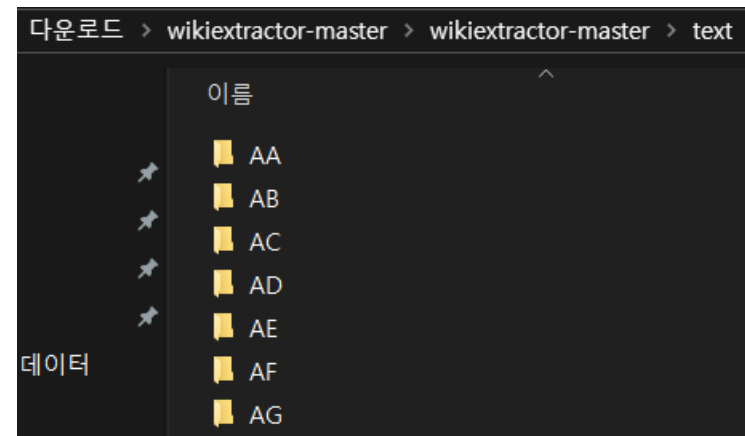


- 그리고 콘솔 창에서 다음의 명령어로 xml을 텍스트 파일로 변환한다
- `python WikiExtractor.py kowiki-latest-pages-articles.xml.bz2`

```
D:\Downloads\wikiextractor-master\wikiextractor-master>python WikiExtractor.py kowiki-latest-pages-articles.xml.bz2
```

- 변환이 완료되면 text 폴더에 AA~AG까지의 폴더가 생성된다

```
명령 프롬프트
INFO:root:2652659   겐저옥현
INFO:root:2652660   찌우타인현 (통안성)
INFO:root:2652674   느느후에현
INFO:root:2652664   느느후아현
INFO:root:2652676   목호아현
INFO:root:2652680   장신초등학교
INFO:root:2652682   연흥현
INFO:root:2652684   연타인현
INFO:root:2652686   투키디데스의 함정
INFO:root:2652687   연쭈현
INFO:root:2652688   타인호아현
INFO:root:2652691   민흥현
INFO:root:2652692   투트어현
INFO:root:2652694   일마누엘 펠라에스
INFO:root:2652696   존 알라스코
INFO:root:2652697   실연, 고마워
INFO:root:2652699   행복의 보호색
INFO:root:2652701   그런 거 아냐
INFO:root:2652709   살바도르 라우렐
INFO:root:2652713   아머
INFO:root:2652717   태초마을
INFO:root:2652723   반스군
INFO:root:2652726   최케빈
INFO:root:2652732   보라 (1982년)
INFO:root:2652738   브랜디 (동음이의)
INFO:root:2652740   라우렐
INFO: Finished 7-process extraction of 486040 articles in 572.9s (848.4 art/s)
INFO: total of page: 835085, total of article page: 486040; total of used article page: 486040
D:\Downloads#wikiextractor-master#wikiextractor-master>
```



변환이 완료된 모습

- 각 폴더에는 wiki\_00, wiki\_01, ... 과 식으로 많은 파일들이 존재한다
- 작업의 편의를 위하여 이 파일들을 모두 하나의 파일로 합친다
- 먼저 cmd에서 다음의 명령어로 각 폴더의 파일들을 하나로 합친다

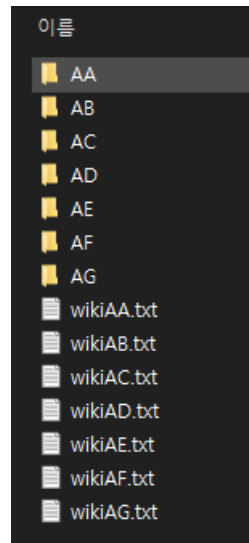
AA 폴더 안에 있는 wiki 로 시작하는 파일(모든 파일)을 wikiAA.txt로 합침

- copy D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki\* wikiAA.txt

```
D:\Downloads\wikiextractor-master\wikiextractor-master\text>copy D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki* wikiAA.txt
```

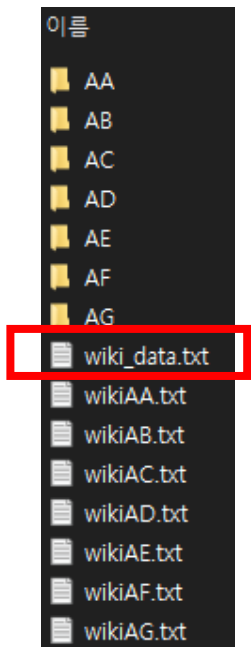
```
D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki_95
D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki_96
D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki_97
D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki_98
D:\Downloads\wikiextractor-master\wikiextractor-master\text\AA\wiki_99
1개 파일이 복사되었습니다.
```

- 이와 같은 식으로 AA ~ AG의 폴더 파일들을 모두 하나로 만든다



- 이제 wikiAA.txt ~ wikiAG.txt의 7개 파일을 다시 하나의 파일로 합친다
- copy D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiA\* wiki\_data.txt

```
D:\Downloads\wikiextractor-master\wikiextractor-master\text>copy D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiA* wiki_data.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAA.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAB.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAC.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAD.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAE.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAF.txt
D:\Downloads\wikiextractor-master\wikiextractor-master\text\wikiAG.txt
1개 파일이 복사되었습니다.
```



# 국립국어원 말뭉치

- 국립국어원 - 자료 - 모두의 말뭉치 - 말뭉치 신청

문화체육관광부  
국립국어원

모두의 말뭉치

최석재님

내정보관리

나가기

말뭉치 신청

사용자 참여

말뭉치 활용

알립니다

인공 지능 언어 능력 평가

말뭉치 신청

미래를 준비하는 소중한 우리말 자원, 말뭉치를 신청하고 신청 내역을 확인할 수 있습니다.

말뭉치 신청

말뭉치 신청 내역

말뭉치 통계

총 32건

찾기

자세히 찾기

- 형태 분석 말뭉치 (형태소 분석 말뭉치 약 300만 어절)
- 문어 말뭉치 (원시 말뭉치 약 10GB)

신규

?

신문 말뭉치 2021

(버전 1.0) 종합지, 전문지, 인터넷 기반

신규

?

국회 회의록 말뭉치 2...

(버전 1.0) 국회 소위원회 회의록

신규

?

추론\_확신성 분석 말...

(버전 1.0) 내포문에서 추출한 가설에

신규

?

맞춤법 교정 말뭉치 2...

(버전 1.0) 온라인에서 나타나는 언어