

# 밑바닥부터 시작하는 딥러닝 ②

파이썬으로 직접 구현하며 배우는 순환신경망과 자연어처리

O'REILLY®

파이썬으로 직접 구현하며 배우는 순환 신경망과 자연어 처리

Deep  
Learning  
from Scratch ②

밑바닥부터 시작하는 딥러닝 2



**1장** 신경망 복습

**2장** 자연어와 단어의 분산 표현

**3장** word2vec

**4장** word2vec 속도 개선

**5장** 순환신경망(RNN)

**6장** 게이트가 추가된 RNN

**7장** RNN을 사용한 문장 생성

**8장** 어텐션

**2장** 자연어와 단어의 분산 표현

--2.1 자연어 처리란

--2.2 시소러스

--2.3 통계 기반 기법

--2.4 통계 기반 기법 개선하기

--2.5 정리

## 자연어와 단어의 분산 표현

마티: “이건 심각한데요heavy.”

박사: “미래에는 이 정도도 무겁단heavy 말이야?”

-영화 <백 투 더 퓨처>\*

이번 장부터는 자연어 처리의 세계로 첫걸음을 내딛게 된다. 자연어 처리가 다루는 분야는 다양하지만, 그 본질적 문제는 **컴퓨터가 우리의 말을 알아듣게(이해하게) 만드는 것**이다. 이번 장은 컴퓨터에 말을 이해시킨다는 것이 무슨 뜻인지, 그리고 어떤 방법들이 존재하는지를 중심으로 다룬다. 특히 **고전적인 기법 (딥러닝 등장 이전의 기법)** 들을 자세히 살펴본다. 딥러닝 기반 기법들은 다음 장에서부터 소개한다. 또한 이번 장은 파이썬으로 텍스트를 다루는 연습도 겸한다. 텍스트를 단어로 분할하는 처리나 단어를 단어 ID로 변환하는 처리 등을 구현한다. 이번 장은 앞으로의 텍스트 처리를 위한 사전 준비라 할 수 있다.

---

# 자연어 처리란

---

## ▪ 단어의 의미

단어는 의미의 최소 단위이다. => 컴퓨터에게 '단어의 의미'를 이해시키는 것이 중요

이번 장에서 살펴볼 컴퓨터에게 '단어의 의미' 이해시키는 방법

- 시소러스를 활용한 기법 (이번 장)
- 통계 기반 기법 (이번 장)
- 추론 기반 기법 (wordvec) (다음 장)

시소러스<sup>thesaurus</sup> (유의어 사전) : 사람이 만든 사전을 이용하는 방법

통계 기반 기법 : 통계 정보로부터 단어를 표현하는 방법

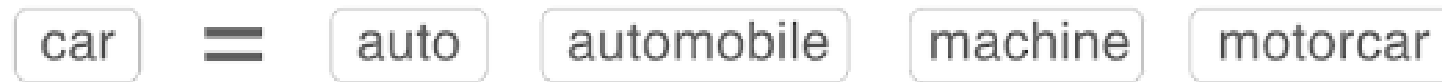
추론 기반 기법 : 신경망을 활용 (word2vec)

# 자연어 처리란

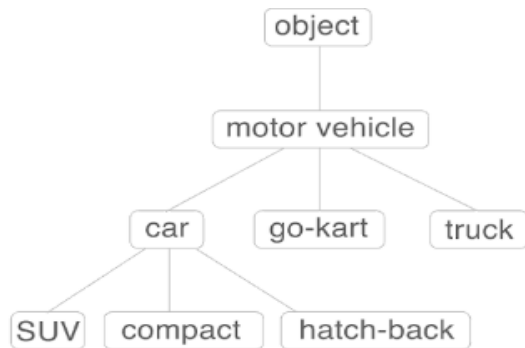
## ■ 시소러스<sup>thesaurus</sup>

‘단어의 의미’를 나타내는 방법으로 사람이 직접 단어의 의미를 정의한 **사전**을 이용하는 방법  
자연어 처리 분야에서는 일반적인 사전이 아닌 **시소러스** 형태의 사전을 애용 (유의어 사전)

[그림 2-1] 동의어의 예



자연어 처리에 이용되는 시소러스에서는 단어 사이의 '상위와 하위' 혹은 '전체와 부분' 등의 관계를 정의해둔 경우가 있다.  
각 단어의 관계를 그래프<sup>graph</sup> 구조로 정의한다.



[그림 2-2] 단어들의 관계로 표현한 그래프

**NOTE\_** 시소러스의 예로 검색 엔진을 생각해 보면 "automobile"과 "car"가 유의어임을 알고 있으면 "car"의 검색 결과에 "automobile"의 검색결과에 포함시켜주면 좋을 것이다.

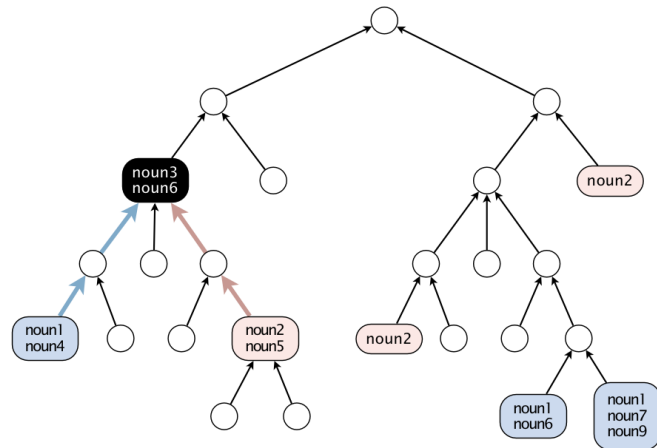
# 자연어 처리란

## ■ 시소러스<sup>thesaurus</sup>

### • WordNet

자연어 처리 분야에서 가장 유명한 시소러스  
프린스턴 대학교에서 1985년부터 구축하기 시작한 전통 있는 시소러스이다.  
지금까지 많은 연구와 다양한 자연어 처리 어플리케이션에서 활용되고 있다.

**WordNet**을 이용하면 유의어를 얻거나 '단어 네트워크'를 이용할 수 있다. 또한 단어 네트워크를 사용하여 단어 사이의 유사도를 구할 수도 있다.



$\text{distance}(\text{noun1}, \text{noun2}) = 4$   
 $\text{sca}(\text{noun1}, \text{noun2}) = \{\text{noun3}, \text{noun6}\}$

# 자연어 처리란

## ■ 시소러스<sup>thesaurus</sup>

### • 시소러스의 문제점

#### ① 시대 변화에 대응하기 어렵다

때때로 새로운 단어가 생겨나고, 옛말은 사라지기도 하며, 시대에 따라 의미가 변하기도 한다.  
이러한 단어의 변화에 대응하려면 시소러스를 사람이 수작업으로 끊임없이 갱신해야 한다.

#### ② 사람을 쓰는 비용은 크다

시소러스를 만드는 데는 엄청난 인적 비용이 발생한다. WordNet에 등록된 단어는 20만개 이상이다.

#### ③ 단어의 미묘한 차이를 표현할 수 없다

시소러스는 뜻이 비슷한 단어들을 묶는다. 그러나 실제로 비슷한 단어들이라도 미묘한 차이는 있는 법이다.  
예로 '빈티지<sup>vintage</sup>'와 '레트로<sup>retro</sup>'는 의미가 같지만, 사용하는 용법은 다르다. 시소러스에서는 이러한 미묘한 차이를 표현할 수 없다.

이처럼 시소러스를 사용하는 기법에는 커다란 3가지의 문제가 있다. 이러한 문제를 피하기 위해 '통계 기반 기법'과 신경망을 사용한 '추론 기반 기법'이 나오게 된다. 이 두 기법에서는 대량의 텍스트 데이터로부터 '단어의 의미'를 자동으로 추출한다.

**NOTE\_** 딥러닝이 실용화되면서 사람이 수작업으로 시소러스나 관계를 설계하던 방식으로부터, 사람의 개입을 최소로 줄이고 텍스트 데이터만으로 원하는 결과를 얻어내는 방향으로 패러다임이 바뀌고 있다.

# 통계 기반 기법

## ▪ 말뭉치<sup>corpus</sup>

이제부터 통계 기반 기법을 살펴보면서 우리는 말뭉치<sup>corpus</sup>를 이용할 것이다.  
말뭉치란 간단히 말하면 대량의 텍스트 데이터이다. 다만 맹목적으로 수집된 데이터가 아닌 자연어 처리 연구나 어플리케이션을 염두에 두고 수집된 텍스트 데이터를 일반적으로 '말뭉치'라고 한다.

말뭉치란 텍스트 데이터에 지나지 않지만, 안에 담긴 문장들은 사람들이 쓴 글이기 때문에,  
사람들이 문장을 쓰는 방법, 단어를 선택하는 방법, 단어의 의미 등의 사람이 알고 있는 자연어에 대한 지식이 담겨 있다.

통계 기반 기법의 목표는 말뭉치에서 **자동**으로, 그리고 효율적으로 **핵심**을 추출하는 것이다.

**WARNING\_** 자연어 처리에 사용되는 말뭉치에는 텍스트 데이터에 추가 정보가 포함되는 경우가 많다.  
예컨대 텍스트 데이터의 단어 각각에 '품사'가 레이블링 될 수 있다. 이럴 경우 말뭉치는 컴퓨터가 다루기 쉬운 형태(트리 구조 등)로 가공되어 주어지는 것이 일반적이다. 이 책에서는 이러한 추가 레이블을 이용하지 않고, 단순한 텍스트 데이터로 주어졌다고 가정한다.

# 통계 기반 기법

## ▪ 말뭉치 전처리하기

자연어 처리에는 다양한 말뭉치가 사용된다. Ex) 위키백과<sup>Wikipedia</sup>, 구글 뉴스<sup>Google News</sup>, 셰익스피어의 작품 등의 텍스트 데이터... 이러한 말뭉치들을 **단어로 분할**①하고, **분할된 단어들을 단어 ID 목록**②으로 만드는 일을 **전처리**라고 한다.

우선 기본적인 문장 하나로 이뤄진 단순한 텍스트를 사용해서 전처리를 해보자.

전처리할 문장

```
>>> text = 'You say goodbye and I say hello.'
```

단어 단위로 분할

```
>>> text = text.lower()
>>> text = text.replace('.', ' .')
>>> text
'you say goodbye and i say hello .'
>>> words = text.split(' ')
>>> words
['you', 'say', 'goodbye', 'and', 'i', 'say', 'hello', '.']
```

**WARNING\_** 여기에선 마침표 앞에 공백을 넣는 임시적인 방법을 적용했지만, 더 현명하고 범용적인 방법으로, '정규표현식<sup>regular expression</sup>'을 이용하는 방법이 있다. re를 impor하고, re.split('(\W+)?', text) 라고 호출하면 단어 단위로 분할할 수 있다.



# 통계 기반 기법

## ▪ 말뭉치 전처리하기

단어를 텍스트 그대로 조작하기란 여러 면에서 불편하므로, 단어에 ID를 부여하고, ID의 리스트로 이용할 수 있도록 한 번 더 손질한다. 이를 위한 자료구조로 파이썬의 딕셔너리(Map)를 이용하여 단어 ID와 단어를 짝지어주는 대응표를 작성한다.

```
>>> word_to_id = {}
>>> id_to_word = {}
>>>
>>> for word in words:
...     if word not in word_to_id: : 중복 불허 처리
...         new_id = len(word_to_id)
...         word_to_id[word] = new_id
...         id_to_word[new_id] = word
```

id\_to\_word : 단어 ID => 단어 < Key : 단어ID, Value : 단어 >

word\_to\_id : 단어 => 단어 ID < Key : 단어, Value : 단어ID >

```
>>> id_to_word
{0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6: '.'}
>>> word_to_id
{'you': 0, 'say': 1, 'goodbye': 2, 'and': 3, 'i': 4, 'hello': 5, '.': 6}
```

# 통계 기반 기법

## ▪ 말뭉치 전처리하기

이처럼 딕셔너리를 사용하면 단어로 단어ID를 검색하거나, 반대로 단어ID를 가지고 단어를 검색할 수 있다.

```
>>> id_to_word[1]
'say'
>>> word_to_id['hello']
5
```

마지막으로 '단어 목록'을 '단어 ID 목록'으로 변경한다. 다음 코드는 파이썬의 리스트 컴프리헨션list comprehension를 사용했다.

```
>>> import numpy as np
>>> corpus = [word_to_id[w] for w in words]
>>> corpus = np.array(corpus)
>>> corpus
array([0, 1, 2, 3, 4, 1, 5, 6])
```

**NOTE\_** 리스트 컴프리헨션이란 리스트나 딕셔너리 등의 반복문 처리를 간단하게 쓰기 위한 파이썬의 기법이다. 예컨대 `xs = [1,2,3,4]`라는 리스트의 각 원소를 제공하고 싶다면 `[x**2 for x in xs]`처럼 쓰면 된다.

# 통계 기반 기법

## ▪ 말뭉치 전처리하기

이상의 처리를 한 데 모아 preprocess()라는 함수로 구현

```
def preprocess(text):
    text = text.lower()
    text = text.replace('.', ' .')
    words = text.split(' ')

    word_to_id = {}
    id_to_word = {}

    for word in words:
        if word not in word_to_id:
            new_id = len(word_to_id)
            word_to_id[word] = new_id
            id_to_word[new_id] = word

    corpus = np.array([word_to_id[w] for w in words])

    return corpus, word_to_id, id_to_word
```

이 함수를 사용하면 다음과 같은 말뭉치 전처리를 수행할 수 있다.

```
>>> text = 'You say goodbye and I say hello.'
>>> corpus, word_to_id, id_to_word = preprocess(text)
```

# 통계 기반 기법

## ▪ 단어의 분산 표현

'색'을 표현하는 방법 중 하나로 RGB라는 세 가지 성분이 어떤 비율로 섞여 있느냐는 방법이 있다. 이러한 비율적인 방법을 3차원의 **벡터로 표현** 가능하다.

여기서 주목할 점은 이러한 RGB 같은 벡터 표현이 색을 더 정확하게 명시할 수 있다는 사실이다.

'색'을 벡터로 표현하듯 간결하고 이치에 맞는 벡터 표현을 단어라는 영역에서도 구축할 수 있다. 이제 우리가 얻고자 하는 것은 '단어의 의미'를 정확하게 파악할 수 있는 벡터 표현이다.

이를 자연어 처리 분야에서는 단어의 **분산표현**distribution representation이라고 한다.

즉, **단어의 벡터화**가 목표이다.

**NOTE\_** 단어의 분산 표현은 단어를 고정 길이의 밀집벡터dense vector로 표현한다. 밀집벡터라 함은 대부분의 원소가 0이 아닌 실수인 벡터를 말한다. 예컨대 3차원의 분산 표현은 [0.21, -0.45, 0.83]과 같은 모습이 된다.

# 통계 기반 기법

## ▪ 분포 가설

자연어 처리에서 단어를 벡터로 표현하는 연구는 수없이 많이 이뤄져 왔는데, 이러한 기법들은 모두 '단어의 의미는 주변 단어에 의해 형성된다' 라는 아이디어에 뿌리를 두고 있다. 이를 분포가설 distributional hypothesis이라 하며, 최근 연구도 대부분이 가설에 기초한다.

분포 가설이 말하고자 하는 바는 매우 간단하다. 단어 자체에는 의미가 없고, 그 단어가 사용된 맥락 context이 의미를 형성한다는 것이다. 여기서 '맥락'이라 함은 주목하는 단어의 주변에 놓인 단어를 가르킨다.

[그림 2-3] 윈도우 크기가 2인 '맥락' (좌우의 두 단어)



you say goodbye and i say hello.

[그림 2-3]처럼 '맥락' 이란 특정 단어를 중심에 둔 그 주변 단어를 말한다. 그리고 맥락의 크기를 '윈도우 크기 window size'라고 한다.

**WARNING.** 여기에서는 좌우로 똑같은 수의 단어를 맥락으로 사용했지만, 상황에 따라서는 왼쪽 단어만 혹은 오른쪽 단어만을 사용하기도 하며, 문장의 시작과 끝을 고려할 수도 있다.

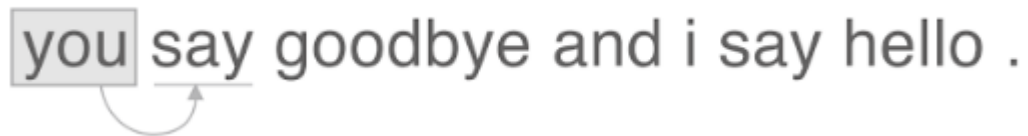
# 통계 기반 기법

## ▪ 동시발생 행렬

분포 가설에 기초해 단어를 벡터로 나타내는 방법을 생각해보면, 주변 단어를 '세어보는' 방법이 자연스럽게 떠오를 것이다. 어떤 단어에 주목했을 때, 그 주변에 어떤 단어가 몇 번이나 등장하는지를 세어 집계하는 방법이다. 이 책에서는 이를 '통계 기반<sup>statistical based</sup>' 기법 이라고 한다.

'You say goodbye and I say hello.' 이 문장을 각 단어의 맥락에 해당하는 단어의 빈도를 세어보자. 윈도우 크기는 1로 하고, 단어 ID가 0인 "you"부터 시작한다.

그림 2-4 단어 "you"의 맥락을 세어본다.



you say goodbye and i say hello .

[그림 2-4]에서 쉽게 알 수 있듯, 단어 "You"의 맥락은 "say"라는 단어 하나뿐이다. 이를 표로 정리하면 다음과 같다.

그림 2-5 단어 "you"의 맥락에 포함되는 단어의 빈도를 표로 정리한다.

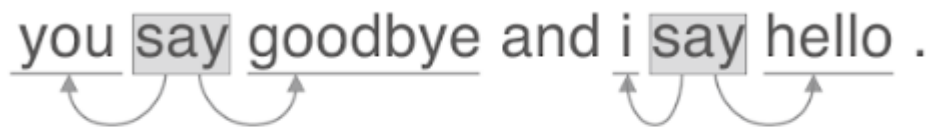
	you	say	goodbye	and	i	hello	.
you	0	1	0	0	0	0	0

# 통계 기반 기법

## ▪ 동시발생 행렬

이번엔 단어 "say"의 맥락에 포함되는 단어의 빈도를 표로 정리하면 다음과 같다.

그림 2-6 단어 "say"의 맥락에 포함되는 단어의 빈도를 표로 정리한다.



	you	say	goodbye	and	i	hello	.
say	1	0	1	0	1	1	0

위의 작업을 모든 단어 (이번 예에서는 총 7개 단어)에 대해서 수행하면 [그림 2-7]과 같다.

[그림 2-7]과 같이 동시발생하는 단어를 표로 정리한 형태가 행렬의 형태를 띤다는 뜻에서 **동시발생 행렬** co-occurrence matrix 이라고 한다.

그림 2-7 모든 단어 각각의 맥락에 해당하는 단어의 빈도를 세어 표로 정리한다.

	you	say	goodbye	and	i	hello	.
you	0	1	0	0	0	0	0
say	1	0	1	0	1	1	0
goodbye	0	1	0	1	0	0	0
and	0	0	1	0	1	0	0
i	0	1	0	1	0	0	0
hello	0	1	0	0	0	0	1
.	0	0	0	0	0	1	0

# 통계 기반 기법

## ▪ 동시발생 행렬

말뭉치로부터 동시발생 행렬을 만들어주는 함수 구현

```
def create_co_matrix(corpus, vocab_size, window_size = 1):
    corpus_size = len(corpus)
    co_matrix = np.zeros((vocab_size, vocab_size), dtype = np.int32)

    for idx, word_id in enumerate(corpus): # 행만큼
        for i in range(1, window_size + 1):
            left_idx = idx - i
            right_idx = idx + i

            if left_idx >= 0:
                left_word_id = corpus[left_idx]
                co_matrix[word_id, left_word_id] += 1

            if right_idx < corpus_size:
                right_word_id = corpus[right_idx]
                co_matrix[word_id, right_word_id] += 1

    return co_matrix
```

(참고) corpus

```
>>> corpus
array([0, 1, 2, 3, 4, 1, 5, 6])
```



# 통계 기반 기법

## ▪ 벡터 간 유사도

앞에서 단어를 벡터로 표현하는 방법을 알아보았다. 그럼 계속해서 벡터 사이의 유사도를 측정하는 방법을 살펴보자. 벡터 사이의 유사도를 측정하는 방법은 다양하다. 대표적으로 벡터의 내적이나 유클리드 거리 등을 꼽을 수 있는데, 단어 벡터의 유사도를 나타낼 때는 **코사인 유사도** cosine similarity를 자주 이용한다. 코사인 유사도는 다음 식으로 정의된다.

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} = \frac{x_1 y_1 + \dots + x_n y_n}{\sqrt{x_1^2 + \dots + x_n^2} \sqrt{y_1^2 + \dots + y_n^2}}$$

분자에는 벡터의 내적, 분모에는 각 벡터의 노름<sup>norm</sup>이 등장한다. Norm은 벡터의 크기를 나타낸 것으로, 여기서는 'L2 norm'을 계산한다.

**NOTE\_** 코사인 유사도를 직관적으로 풀어보자면 '두 벡터가 가르키는 방향이 얼마나 비슷한가'이다. 두 벡터의 방향이 완전히 같다면 코사인 유사도가 1이 되며, 완전히 반대라면 -1이 된다.

# 통계 기반 기법

## ▪ 벡터 간 유사도

코사인 유사도를 파이썬 함수로 구현

```
def cos_similarity(x, y):  
    nx = x / np.sqrt(np.sum(x**2)) # x의 정규화  
    ny = y / np.sqrt(np.sum(y**2)) # y의 정규화  
    return np.dot(nx, ny)
```

이 코드에서 x와 y는 넘파이 배열이라고 가정한다. 위의 코드로도 코사인 유사도를 구할 수 있지만, 만약 x나 y가 제로 벡터(위소가 모두 0인 벡터)가 들어오면 **'divide by zero'** 오류가 발생해버린다.

이 문제를 해결하는 전통적인 방법으로, 나눌 때 분모에 아주 작은 값을 더해주는 방법이 있다. 작은 값을 뜻하는 eps(epsilon)를 인수로 받도록 하고, 이 인수의 값을 지정하지 않으면 default 값으로  $1e-8(=0.00000001)$ 이 설정되도록 수정한다.

```
def cos_similarity(x, y, eps = 1e-8):  
    nx = x / np.sqrt(np.sum(x**2) + eps) # x의 정규화  
    ny = y / np.sqrt(np.sum(y**2) + eps) # y의 정규화  
    return np.dot(nx, ny)
```

**NOTE\_** 여기에서 eps 값으로  $1e-8$ 을 사용했는데, 이 정도 작은 값이면 일반적으로 부동소수점 계산 시 '반올림'되어 다른 값에 흡수된다. 그렇기 때문에 eps를 더한다고 해서 최종 계산 결과에는 영향을 주지 않는다.

# 통계 기반 기법

## ▪ 벡터 간 유사도

앞의 함수를 사용하여 벡터의 유사도를 구하면 다음과 같다. 다음은 "you"와 "i"의 유사도를 구하는 코드이다.

```
text = "You say goodbye and i say hello."
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)

c0 = C[word_to_id['you']] # "you"의 단어 벡터
c1 = C[word_to_id['i']]   # "i"의 단어 벡터
print(cos_similarity(c0, c1))
# 0.7071067691154799
```

실행 결과 코사인 유사도는 0.70...으로 나왔다. 코사인 유사도 값은 -1에서 1 사이이므로, 이 값은 비교적 높다 (유사성이 크다)고 말할 수 있다.

# 통계 기반 기법

## ▪ 유사 단어의 랭킹 표시

코사인 유사도까지 구현을 했으니, 이 함수를 활용해서 유용한 기능을 구현해보자.  
어떠한 단어가 검색어로 주어지면, 그 검색어와 비슷한 단어를 **유사도 순으로 출력하는 함수**를 만들어보자.

```
most_similar(query, word_to_id, id_to_word, word_matrix, top = 5)
```

인수명	설명
query	검색어(단어)
word_to_id	단어에서 단어 ID로의 딕셔너리
id_to_word	단어 ID에서 단어로의 딕셔너리
word_matrix	단어 벡터들을 한데 모은 행렬.
top	상위 몇 개까지 출력할지 설정

# 통계 기반 기법

## ▪ 유사 단어의 랭킹 표시

```
most_similar(query, word_to_id, id_to_word, word_matrix, top = 5)
```

```
def most_similar(query, word_to_id, id_to_word, word_matrix, top=5):
    # 1. 검색어를 꺼낸다.
    if query not in word_to_id:
        print('%s(을)를 찾을 수 없습니다.' % query)
        return

    print('\n[query] ' + query)
    query_id = word_to_id[query]
    query_vec = word_matrix[query_id]

    # 2. 코사인 유사도 계산
    vocab_size = len(id_to_word)

    similarity = np.zeros(vocab_size)
    for i in range(vocab_size):
        similarity[i] = cos_similarity(word_matrix[i], query_vec)

    # 3. 코사인 유사도를 기준으로 내림차순으로 출력
    count = 0
    for i in (-1 * similarity).argsort():
        if id_to_word[i] == query:
            continue
        print(' %s: %s' % (id_to_word[i], similarity[i]))

        count += 1
        if count >= top:
            return
```

이 코드는 다음 순서로 동작한다.

- ① 검색어의 단어 벡터를 꺼낸다.
- ② 검색어의 단어 벡터와 다른 모든 단어 벡터와의 코사인 유사도를 각각 계산
- ③ 계산한 코사인 유사도 결과를 기준으로 값이 높은 순서대로 출력

③에서는 similarity 배열에 담긴 워드의 인덱스를 내림차순으로 정렬한 후 상위 원소들을 출력한다. 이때 배열 index의 정렬을 바꾸는 데는 argsort() 메서드를 사용한다. argsort()는 넘파이 배열의 원소를 오름차순으로 정렬한다. (단 반환 값은 배열의 index)

argsort()는 배열을 오름차순으로 정렬하는데 우리가 필요한 것은 내림차순이므로 Similarity에 -1을 곱해서 sorting 한다.

# 통계 기반 기법

## ▪ 유사 단어의 랭킹 표시

앞에서 정의한 함수 사용해보자. "you"를 검색어로 지정해 유사한 단어들을 출력해보자.

```
text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)

most_similar('you', word_to_id, id_to_word, C, top=5)
```

이 코드를 실행하면 다음 결과를 얻을 수 있다.

```
[query] you
goodbye: 0.7071067811865475
i: 0.7071067811865475
hello: 0.7071067811865475
say: 0.0
and: 0.0
```

이 결과는 검색어 "you"와 유사한 단어를 상위 5개만 출력한 것이다. "you"와 "I"가 유사도가 높다는 것은 납득이 되지만, "goodbye"와 "hello"의 유사도가 높다는 것은 우리의 직관과는 거리가 멀다. 지금은 말뭉치의 크기가 너무 작다는 것이 원인이다.

# 통계 기반 기법 개선하기

## ▪ 상호정보량

앞 절에서 본 동시발생 행렬의 원소는 두 단어가 동시에 발생한 횟수를 나타낸다.  
그러나 이 '발생 횟수'라는 것은 사실 그리 좋은 특징이 아니다.  
고빈도 단어 (많이 출현하는 단어)를 고려해보면 그 이유를 알 수 있다.

예컨데 말뭉치에서 "the"와 "car"의 동시발생을 생각해보면, 동시발생 횟수는 아주 많을 것이다.  
한편 "car"와 "drive"는 확실히 관련이 깊다.  
하지만 단순히 등장 횟수만을 본다면 "car"는 "drive" 보다는 "the"와 관련성이 크다고 나올 것이다.  
이는 "the"가 고빈도 단어이기 때문에 등장 횟수가 많을 수밖에 없는 특징 때문에 그렇다.

이 문제를 해결하기 위해 **점별 상호정보량** Pointwise Mutual Information (**PMI**) 라는 척도를 사용한다.

$$\text{PMI}(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)}$$

$P(x)$ 는  $x$ 가 일어날 확률,  $P(y)$ 는  $y$ 가 일어날 확률,  $P(x,y)$ 는  $x$ 와  $y$ 가 동시에 일어날 확률을 뜻한다.  
이 PMI 값이 높을수록 관련성이 높다는 의미이다.

=> 즉 단순히  $P(x,y)$ 만을 고려하는 방법에서,  $P(x)$ 와  $P(y)$  값들까지 고려해서 수치화 시킨 값이다.

$$\text{PMI}(\text{"the"}, \text{"car"}) \approx 2.32, \text{PMI}(\text{"car"}, \text{"drive"}) \approx 7.97$$

"the"와 "car"와 "drive"가 각각 1000번, 20번, 10번 등장했고, "the" & "car"의 동시발생 횟수는 10, "car" & "drive" 동시발생 횟수는 5회라고 가정

# 통계 기반 기법 개선하기

## ▪ 상호정보량

앞의 계산 결과로  $\text{PMI}(\text{"the"}, \text{"car"}) \approx 2.32$ ,  $\text{PMI}(\text{"car"}, \text{"drive"}) \approx 7.97$  라는 결과가 나왔다.  
PMI를 이용하면 "car"는 "the"보다 "drive"와 관련성이 강해진다.  
이러한 결과는 단독으로 출현하는 횟수가 고려되었기 때문이다.

이제 PMI라는 척도를 얻었지만, 두 단어의 동시발생 횟수가 0이면  $\log_2 0 = -\infty$  가 된다는 문제가 생긴다.  
이러한 문제를 피하기 위해 실제로는 **양의 상호정보량**  $\text{Positive PMI (PPMI)}$ 을 사용한다.

$$\text{PPMI}(x,y) = \max(0, \text{PMI}(x,y))$$

이 식에 따라 PMI가 음수일 때는 0으로 취급한다.

이제 단어 사이의 관련성을 0 이상의 실수로 나타낼 수 있다.



# 통계 기반 기법 개선하기

## ▪ 상호정보량

PPMI 구현

```
def ppmi(C, verbose=False, eps = 1e-8):  
  
    M = np.zeros_like(C, dtype=np.float32)  
    N = np.sum(C)  
    S = np.sum(C, axis=0)  
    total = C.shape[0] * C.shape[1]  
    cnt = 0  
  
    for i in range(C.shape[0]):  
        for j in range(C.shape[1]):  
            pmi = np.log2(C[i, j] * N / (S[j]*S[i]) + eps)  
            M[i, j] = max(0, pmi)  
  
            if verbose:  
                cnt += 1  
                if cnt % (total//100) == 0:  
                    print('%.1f%% 완료' % (100*cnt/total))  
  
    return M
```

인수 **C**는 동시발생 행렬, **verbose**는 진행상화 출력 여부를 결정하는 플래그이다.  
그리고 앞서와 같이 **divide by zero** 에러를 피하기 위해 **eps**라는 작은 값을 사용했다.

# 통계 기반 기법 개선하기

## 상호정보량

동시발생 행렬을 PPMI 행렬로 변환

```
import numpy as np

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)
W = ppmi(C)

np.set_printoptions(precision=3) # 유효 자릿수를 세 자리로 표시
print('동시발생 행렬')
print(C)
print('-'*50)
print('PPMI')
print(W)
```

### OUTPUT

동시발생 행렬

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

PPMI

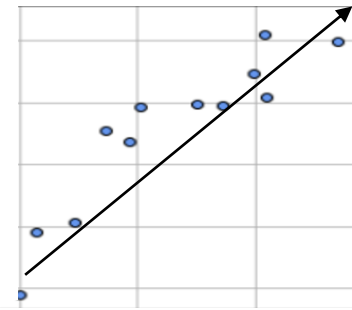
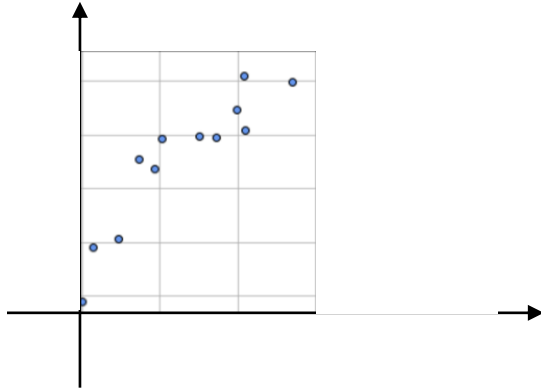
```
[[0.    1.807 0.    0.    0.    0.    0.   ]
 [1.807 0.    0.807 0.    0.807 0.807 0.   ]
 [0.    0.807 0.    1.807 0.    0.    0.   ]
 [0.    0.    1.807 0.    1.807 0.    0.   ]
 [0.    0.807 0.    1.807 0.    0.    0.   ]
 [0.    0.807 0.    0.    0.    0.    2.807]
 [0.    0.    0.    0.    0.    2.807 0.   ]]
```

그러나 PPMI 행렬에도 여전히 큰 문제가 있다. 말뭉치의 어휘 수가 증가함에 따라 각 단어 벡터의 차원수도 증가한다는 문제다. 예를 들어 어휘 수가 10만 개라면 그 벡터의 차원 수도 10만이 된다. 위의 행렬을 보면 원소 대부분이 0인 **희소행렬** sparse matrix임을 알 수 있다. 이를 이용하여 문제에 대처하고자 자주 수행하는 기법이 바로 **벡터의 차원 감소**이다.

# 통계 기반 기법 개선하기

## ▪ 차원 감소

**차원 감소** dimensionality reduction는 문자 그대로 벡터의 차원을 줄이는 방법이다. 그러나 단순히 줄이기만 하는게 아니라, '중요한 정보'는 최대한 유지하면서 줄이는 게 핵심이다. 직관적인 예로, [그림 2-8]처럼 데이터의 분포를 고려해 중요한 '축'을 찾는 일을 수행한다.



[그림 2-8] 그림으로 이해하는 차원 감소: 2차원 데이터를 1차원으로 표현하기 위해 중요한 축(데이터를 넓게 분포시키는 축)을 찾는다.

**NOTE\_** 원소 대부분이 0인 행렬 또는 벡터를 '희소행렬 sparse matrix' 또는 '희소벡터 sparse vector'라 한다. 차원 감소의 핵심을 희소벡터에서 중요한 축을 찾아내어 더 작은 차원으로 다시 표현하는 것인데, 차원감소의 결과는 원소 대부분이 0이 아닌 값으로 구성된 '밀집벡터'로 변환된다. 이 조밀한 벡터야말로 우리가 원하는 단어의 분산 표현이다.

# 통계 기반 기법 개선하기

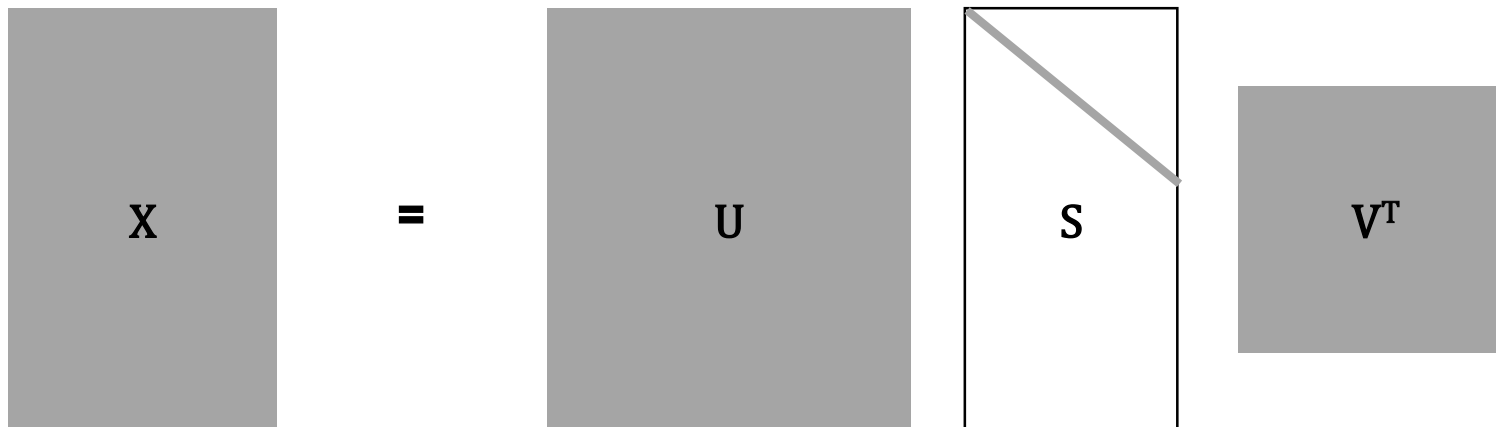
## ▪ 차원 감소

차원을 감소시키는 방법은 여러가지인데, 우리는 **특잇값분해** Singular Value Decomposition(**SVD**)를 이용한다.  
SVD는 임의의 행렬을 세 행렬의 곱으로 분해하며, 수식으로는 다음과 같다.

$$X = USV^T$$

위의 식과 같이 SVD는 임의의 행렬  $X$ 를  $U, S, V$ 라는 세 행렬의 곱으로 분해한다.  
여기서  $U$ 와  $V$ 는 직교행렬(orthogonal matrix)이고, 그 열벡터는 서로 직교한다. 또한  $S$ 는 대각행렬(diagonal matrix) (대각성분 외는 모두 0)이다.  
이 수식을 시각적으로 표현한 것이 [그림 2-9]이다.

[그림 2-9] SVD에 의한 행렬의 변환 (행렬의 '흰 부분'은 원소가 0임을 뜻함)



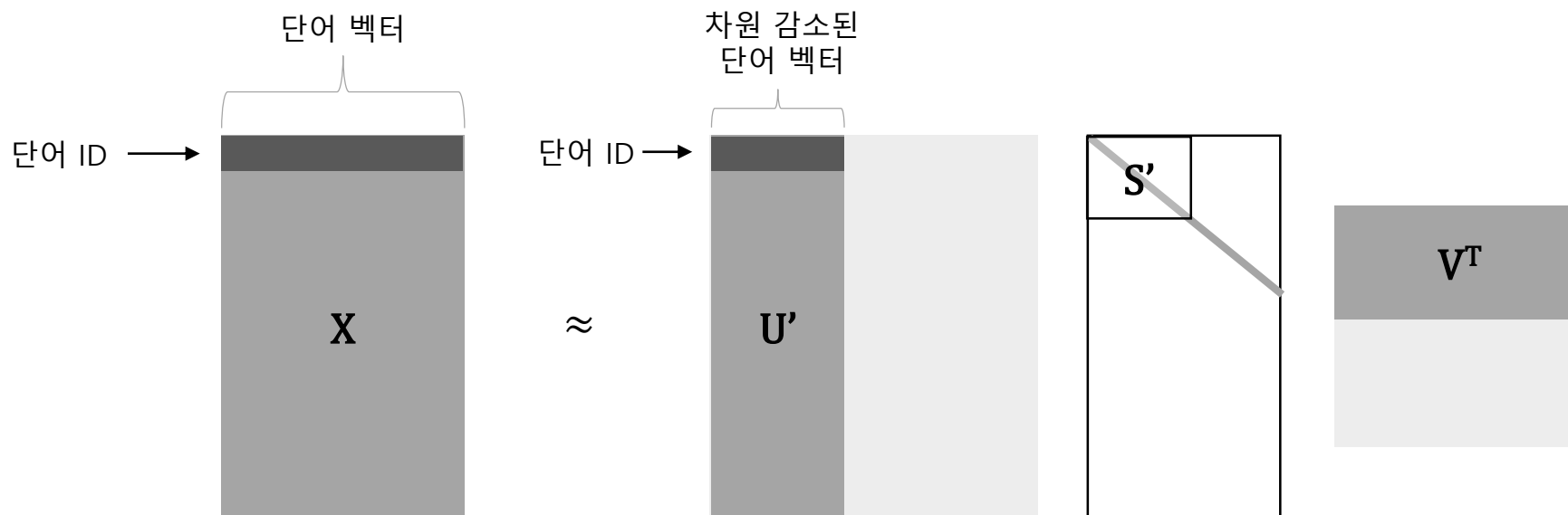
# 통계 기반 기법 개선하기

## 차원 감소

SVD에서  $U$ 는 직교행렬이다. 그리고 이 직교행렬은 어떠한 공간의 축(기저)을 형성한다.

지금 우리의 맥락에서는 이  $U$  행렬을 '단어 공간'으로 취급할 수 있다. 또한  $S$ 는 대각행렬로, 그 대각성분에는 '특잇값(Singular value)'이 큰 순서로 나열되어 있다. 특잇값이란, 쉽게 말해 '해당 축'의 중요도라고 간주할 수 있다. 그래서 앞의 [그림 2-9]는 [그림 2-10]과 같이 **중요도가 높은 원소들만을 뽑아내는 방법**이라고 할 수 있다.

[그림 2-10] SVD에 의한 차원 감소



# 통계 기반 기법 개선하기

## ▪ 차원 감소

SVD에 의한 차원 감소

```
import numpy as np

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(id_to_word)
C = create_co_matrix(corpus, vocab_size, window_size=1)
W = ppmi(C)

# SVD
U, S, V = np.linalg.svd(W)
```

이 코드에서 SVD에 변환된 밀집벡터 표현은 변수 U에 저장된다.

```
np.set_printoptions(precision=3) # 유효 자릿수를 세 자리로 표시
print(C[0])
# [0 1 0 0 0 0 0]
print(W[0])
# [0.    1.807 0.    0.    0.    0.    0.    ]
print(U[0])
# [-3.409e-01 -1.110e-16 -3.886e-16 -1.205e-01  0.000e+00  9.323e-01
#  2.226e-16]
```

밀집벡터의 차원을 감소시키려면, 예컨대 차원 벡터로 줄이려면 단순히 처음의 두 원소를 꺼내면 된다.

# 통계 기반 기법 개선하기

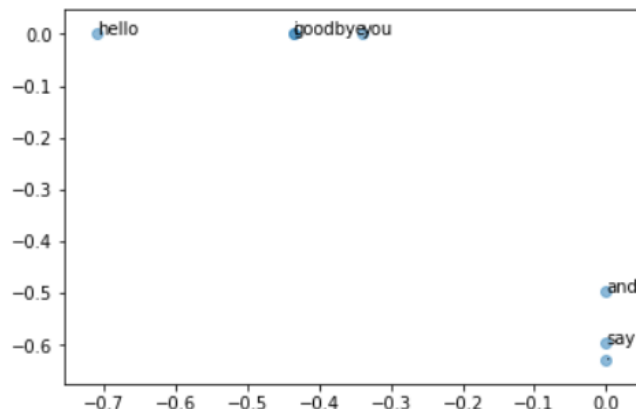
## 차원 감소

밀집벡터의 차원을 감소시키려면, 예컨대 차원 벡터로 줄이려면 단순히 처음의 두 원소를 꺼내면 된다.

```
print(U[0, :2])  
# [-3.409e-01 -1.110e-16]
```

이제 각 단어를 2차원 벡터로 표현한 후 그래프로 그려보자.

```
import matplotlib.pyplot as plt  
  
for word, word_id in word_to_id.items():  
    plt.annotate(word, (U[word_id, 0], U[word_id, 1]))  
plt.scatter(U[:,0], U[:,1], alpha=0.5)  
plt.show()
```



[그림 2-11]을 보면 "goodbye"와 "hello", "you"와 "I"가 제법 가깝게 있음을 알 수 있다. 하지만 지금 사용한 말뭉치가 아주 작아서 이 결과는 석연치 않다. 이제는 PTB 데이터셋이라는 더 큰 말뭉치를 사용하여 똑같은 실험을 수행해보자.

**WARNING\_** SVD 계산은  $O(N^3)$ 이 걸린다. 이는 현실적으로 감당하기 어려운 수준이므로, 보통 Truncated SVD 같은 더빠른 기법을 이용한다.

[그림 2-11] 동시발생 행렬에 SVD를 적용한 후 각 단어를 2차원 벡터로 변환해 그린 그래프

# 통계 기반 기법 개선하기

## ▪ PTB 데이터셋

이제 본격적으로 적당한 말뭉치들을 이용해보자. 그 주인공이 바로 펜 트리뱅크 Penn Treebank(PTB)이다.

**NOTE\_** PTB 말뭉치는 주어진 기법의 품질을 측정하는 벤치마크로 자주 이용된다.

PTB는 텍스트 파일로 제공되며, 원래의 PTB 문장에 몇가지 전처리를 해두었다.  
예컨데 희소한 단어를 <unk>라는 특수 문자로 치환한다거나 구체적인 숫자를 "N"으로 대체하는 등의 작업이 적용되었다.

[그림 2-12] PTB 말뭉치(텍스트 파일)의 예

```
the fed arranged $ N billion of customer repurchase agreements tuesday the
second repurchase agreement in two days the move which <unk> capital into
the system is seen as an effort to <unk> the <unk> markets that the u.s.
central bank is ready to provide the ample liquidity but other analysts
contend that while the fed 's move to loosen credit has n't been
aggressive it nevertheless sends a clear signal that at least for now the
fed has <unk> its grip on credit they add that the fed has allowed the key
federal funds interest rate to dip to about N N N from its levels of just
```



# 통계 기반 기법 개선하기

## ▪ PTB 데이터셋

이제 PTB 데이터셋에 통계 기반 기법을 적용해보자. 이번에는 큰 행렬에 SVD를 적용해야 하므로 고속 SVD를 이용하였다. 고속 SVD는 sklearn 모듈의 randomized\_svd()를 사용하였다.

```
import numpy as np
from dataset import ptb
from sklearn.utils.extmath import randomized_svd

window_size = 2
wordvec_size = 100

corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)
print('동시발생 수 계산 ...')
C = create_co_matrix(corpus, vocab_size, window_size)
print('PPMI 계산 ...')
W = ppmi(C, verbose=True)

print('calculating SVD ...')
# truncated SVD (빠르다!)
U, S, V = randomized_svd(W, n_components=wordvec_size,
                          n_iter=5, random_state=None)

word_vecs = U[:, :wordvec_size]

querys = ['you', 'year', 'car', 'toyota']
for query in querys:
    most_similar(query, word_to_id,
                  id_to_word, word_vecs, top=5)
```

### OUTPUT

```
[query] you
i: 0.702039909619
we: 0.699448543998
`ve: 0.554828709147
do: 0.534370693098
else: 0.512044146526

[query] year
month: 0.731561990308
quarter: 0.658233992457
last: 0.622425716735
earlier: 0.607752074689
next: 0.601592506413

[query] car
luxury: 0.620933665528
auto: 0.615559874277
cars: 0.569818364381
vehicle: 0.498166879744
corsica: 0.472616831915

[query] toyota
motor: 0.738666107068
nissan: 0.677577542584
motors: 0.647163210589
honda: 0.628862370943
lexus: 0.604740429865
```

## 이번 장에서 배운 내용

- WordNet 등의 시소러스를 이용하면 유의어를 얻거나 단어 사이의 유사도를 측정하는 등 유용한 작업을 할 수 있다.
- 시소러스 기반 기법은 시소러스를 작성하는 데 엄청난 인적 자원이 든다거나 새로운 단어에 대응하기 어렵다는 문제가 있다.
- 현재는 말뭉치를 이용해 단어를 벡터화하는 방식이 주로 쓰인다.
- 최근의 단어 벡터화 기법들은 대부분 '단어의 의미는 주변 단어에 의해 형성된다'는 분포 가설에 기초한다.
- 통계 기반 기법은 말뭉치 안의 각 단어에 대해서 그 단어의 주변 단어의 빈도를 집계한다. (동시발생 행렬)
- 동시발생 행렬을 PPMI 행렬로 반환하고 다시 차원을 감소시킴으로써, 거대한 '희소벡터'를 작은 '밀집벡터'로 변환할 수 있다.
- 단어의 벡터 공간에서는 의미가 가까운 단어는 그 거리도 가까울 것으로 기대된다.