Word Senses: Similarity and Ambiguity

Figure 1: Philip Resnik

Word Sense from Thesaurus

이전 섹션에서, 단어는 내부에 의미를 지니고 있고 그 의미는 개념과 같아서 계층적 구조를 지닌다고 하였습니다. 만약 그 계층구조를 잘 분석하고 분류하여 데이터베이스로 구축한다면, 우리가 자연어처리를 하고자 할 때 매우 큰 도움이 될 것입니다. 이런 용도로 구축된 데이터베이스를 어휘분류사전(thesaurus)라고 부릅니다. 이번 섹션에서는 thesaurus의 대표인 WordNet에 대해 다루어 보겠습니다.

WordNet

WordNet(워드넷)은 1985년부터 심리학 교수인 George Armitage Miller 교수의 지도하에 프린스턴 대학에서 만든 프로그램 입니다. 처음에는 주로

기계번역(Machine Translation)을 돕기 위한 목적으로 만들어졌으며, 따라서 동의어 집합(Synset) 또는 상위어(Hypernym)나 하위어(Hyponym)에 대한 정보가 특히 잘 구축되어 있는 것이 장점 입니다. 단어에 대한 상위어와 하위어 정보를 구축하게 됨으로써, Directed Acyclic Graph(유방향 비순환 그래프)를 이루게 됩니다. (Tree구조가 아닌 이유는 하나의 노드가 여러 상위 노드를 가질 수 있기 때문입니다.)

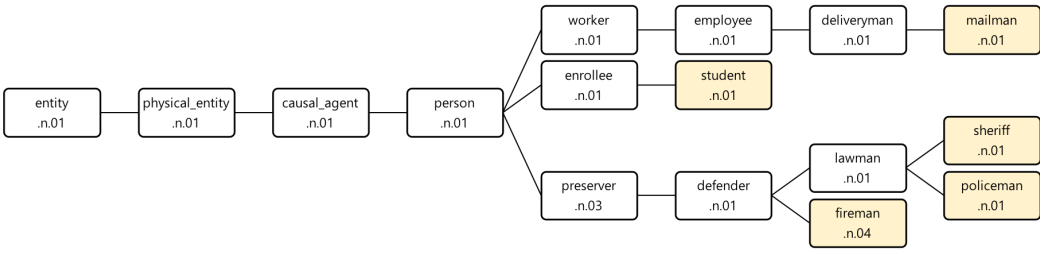
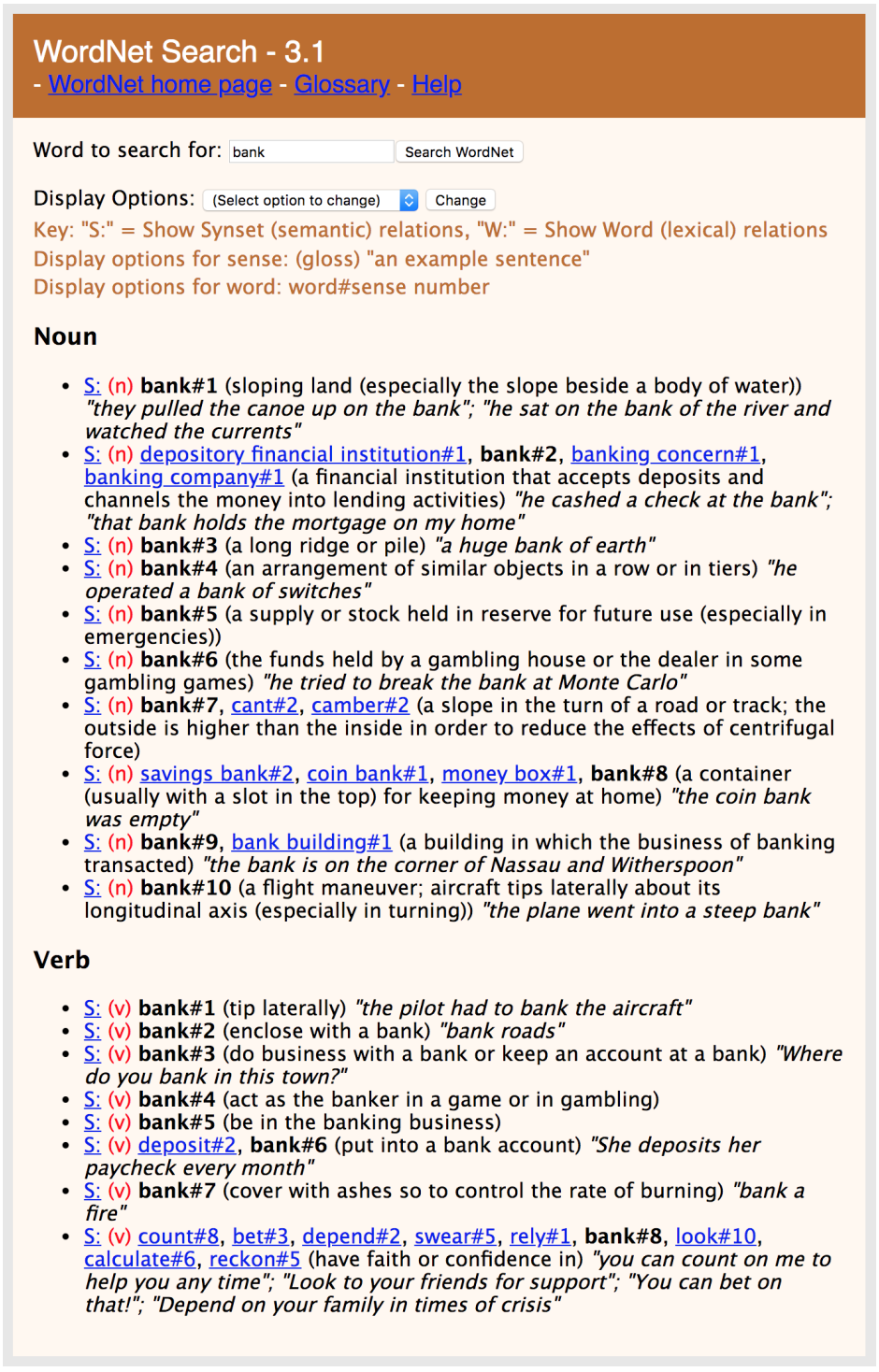


Figure 2: 각 단어별 top-1 sense의 top-1 hypernym만 선택하여 tree로 나타낸 경우

WordNet은 프로그램으로 제공되므로 다운로드 받아 설치할 수도 있고, 웹사이트에서 바로 이용 할 수도 있습니다. 또한, NLTK에 랩핑(wrapping)되어 포함되어 있어, import하여 사용 가능합니다. 아래는 WordNet 웹사이트에서 bank를 검색한 결과 입니다.

그림에서와 같이 bank라는 단어에 대해서 명사(noun)일때의 의미 10개, 동사(verb)인 경우의 의미 8개를 정의 해 놓았습니다. 명사 bank#2의 경우에는 여러 다른 표현(depository finaancial institution#1, banking concern#1)들도 같이 게시되어 있는데, 이것은

이처럼 WordNet은 단어 별 여러가지 가능한 의미를 미리 정의 하고, numbering 해 놓았습니다. 또한 각 의미별로 비슷한 의미를 갖는 동의어(Synonym)를 링크 해 놓아, Synset을 제공합니다. 이것은 단어 중의성 해소에 있어서 매우 좋은 labeled data가 될 수 있습니다. 만약 WordNet이 없다면, 각 단어별로 몇 개의 의미가 있는지조차 알 수가 없을 것입니다. 죽, WordNet 덕분에 우리는 이 데이터를 바탕으로 supervised learning(지도 학습)을 통해 단어 중의성 해소 문제를 풀 수 있습니다.



WordNet Search - 3.1

- WordNet home page - Glossary - Help

Word to search for: bank Search WordNet

Display Options: (Select option to change) Change

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations

Display options for sense: (gloss) "an example sentence"

Display options for word: word#sense number

Noun

- S: (n) **bank#1** (sloping land (especially the slope beside a body of water)) *"they pulled the canoe up on the bank"; "he sat on the bank of the river and watched the currents"*
- S: (n) depository financial institution#1, **bank#2**, banking concern#1, banking company#1 (a financial institution that accepts deposits and channels the money into lending activities) *"he cashed a check at the bank"; "that bank holds the mortgage on my home"*
- S: (n) **bank#3** (a long ridge or pile) *"a huge bank of earth"*
- S: (n) **bank#4** (an arrangement of similar objects in a row or in tiers) *"he operated a bank of switches"*
- S: (n) **bank#5** (a supply or stock held in reserve for future use (especially in emergencies))
- S: (n) **bank#6** (the funds held by a gambling house or the dealer in some gambling games) *"he tried to break the bank at Monte Carlo"*
- S: (n) **bank#7**, cant#2, camber#2 (a slope in the turn of a road or track; the outside is higher than the inside in order to reduce the effects of centrifugal force)
- S: (n) savings bank#2, coin bank#1, money box#1, **bank#8** (a container (usually with a slot in the top) for keeping money at home) *"the coin bank was empty"*
- S: (n) **bank#9**, bank building#1 (a building in which the business of banking transacted) *"the bank is on the corner of Nassau and Witherspoon"*
- S: (n) **bank#10** (a flight maneuver; aircraft tips laterally about its longitudinal axis (especially in turning)) *"the plane went into a steep bank"*

Verb

- S: (v) **bank#1** (tip laterally) *"the pilot had to bank the aircraft"*
- S: (v) **bank#2** (enclose with a bank) *"bank roads"*
- S: (v) **bank#3** (do business with a bank or keep an account at a bank) *"Where do you bank in this town?"*
- S: (v) **bank#4** (act as the banker in a game or in gambling)
- S: (v) **bank#5** (be in the banking business)
- S: (v) deposit#2, **bank#6** (put into a bank account) *"She deposits her paycheck every month"*
- S: (v) **bank#7** (cover with ashes so to control the rate of burning) *"bank a fire"*
- S: (v) count#8, bet#3, depend#2, swear#5, rely#1, **bank#8**, look#10, calculate#6, reckon#5 (have faith or confidence in) *"you can count on me to help you any time"; "Look to your friends for support"; "You can bet on that!"; "Depend on your family in times of crisis"*

Figure 3: WordNet 웹사이트에서 단어 'bank'를 검색 한 결과

한국어 WordNet

다행히 영어 WordNet과 같이 한국어를 위한 WordNet도 존재 합니다. 다만, 아직까지 표준이라고 할만큼 정해진 것은 없고 몇 개의 WordNet이 존재하고 있습니다. 아직까지 지속적으로 발전하고 있는 만큼, 작업에 따라 필요한 한국어 WordNet을 이용하면 좋습니다.

이름	기관	웹사이트
KorLex	부산대학교	http://korlex.pusan.ac.kr/
Korean WordNet(KWN)	KAIST	http://wordnet.kaist.ac.kr/

Word Similarity Based on Path in WordNet

```
from nltk.corpus import wordnet as wn

def get_hypernyms(synset):
    current_node = synset
    while True:
        print(current_node)
        hypernym = current_node.hypernyms()
        if len(hypernym) == 0:
            break
        current_node = hypernym[0]
```

위의 코드를 사용하면 WordNet에서 특정 단어의 최상위 부모 노드까지의 경로를 구할 수 있습니다. 아래와 같이 'policeman'은 'firefighter', 'sheriff'와 매우 비슷한 경로를 가짐을 알 수 있습니다. 'student'와도 매우 비슷하지만, 'mailman'과 좀 더 비슷함을 알 수 있습니다.

```
>>> get_hypernyms(wn.synsets('policeman')[0])
Synset('policeman.n.01')
Synset('lawman.n.01')
Synset('defender.n.01')
Synset('preserver.n.03')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
```

```
Synset('entity.n.01')
```

```
>>> get_hypernyms(wn.synsets('firefighter')[0])
Synset('fireman.n.04')
Synset('defender.n.01')
Synset('preserver.n.03')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')
```

```
>>> get_hypernyms(wn.synsets('sheriff')[0])
Synset('sheriff.n.01')
Synset('lawman.n.01')
Synset('defender.n.01')
Synset('preserver.n.03')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')
```

```
>>> get_hypernyms(wn.synsets('mailman')[0])
Synset('mailman.n.01')
Synset('deliveryman.n.01')
Synset('employee.n.01')
Synset('worker.n.01')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')
```

```
>>> get_hypernyms(wn.synsets('student')[0])
Synset('student.n.01')
Synset('enrollee.n.01')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')
```

위로 부터 얻어낸 정보들을 취합하여 그래프로 나타내면 아래와 같습니다. 그림에서

각 leaf 노드들은 코드에서 쿼리로 주어진 단어들이 됩니다.

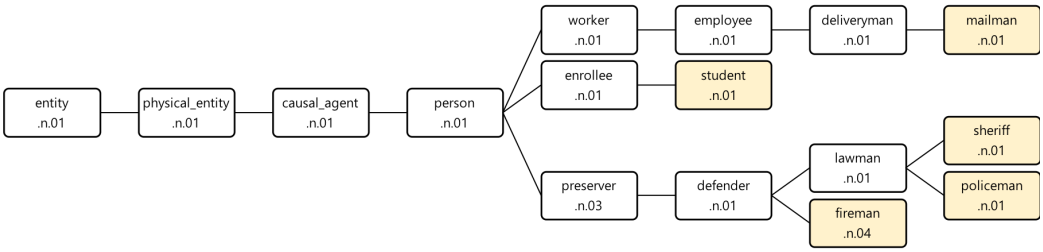


Figure 4: 각 단어들의 쿼리 결과 구조도

이때 각 노드들간에 거리를 우리는 구할 수 있습니다. 아래의 그림에 따르면 student에서 fireman으로 가는 최단거리에는 enrollee, person, preserver, defender 노드들이 위치하고 있습니다. 따라서 student와 fireman의 거리는 5임을 알 수 있습니다.

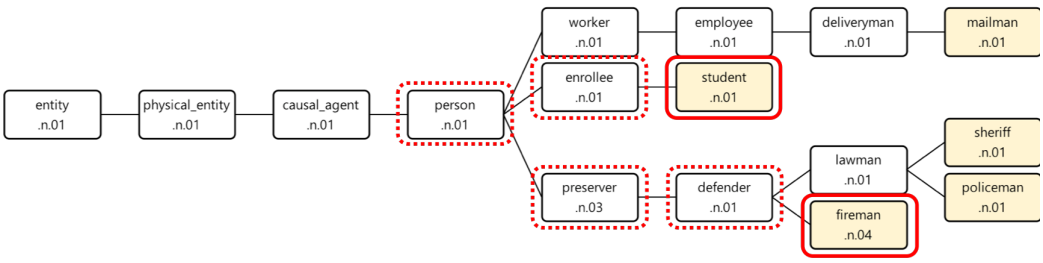


Figure 5: student와 fireman 사이에 위치한 노드들(점선)

이처럼 우리는 각 leaf 노드 간의 최단 거리를 알 수 있고, 이것을 유사도로 치환하여 활용할 수 있습니다. 당연히 거리가 멀수록 단어간의 유사도는 떨어질테니, 아래와 같은 공식을 적용 해 볼 수 있습니다.

$$\text{similarity}(w, w') = -\log \text{distance}(w, w')$$

위와 같이 사전(thesaurus) 기반의 정보를 활용하여 단어간의 유사도를 구할 수 있습니다. 하지만 사전을 구축하는데는 너무 많은 비용과 시간이 소요 됩니다. 또한, 아무 사전이나 되는 것이 아닌, hypernym과 hyponym이 잘 반영되어 있어야 할 것입니다. 이처럼, 사전에 기반한 유사도를 구하는 방식은 비교적 정확한 값을 구할 수 있으나, 한계가 뚜렷합니다. # Appendix: TF-IDF

이번 섹션에서는 텍스트 마이닝(Text Mining)에서 중요하게 사용되는 TF-IDF를 다뤄보겠습니다. TF-IDF는 어떤 단어 w가 문서 d 내에서 얼마나 중요한지 나타내는 수치 입니다. 이 수치가 높을 수록 w는 d를 대표하는 성격을 띄게 된다고 볼 수도 있습니다.

TF-IDF의 TF는 Term Frequency로 단어의 문서 내에 출현한 횟수를 의미 합니다. 그리고 IDF는 Inverse Document Frequency로 그 단어가 출현한 문서의 숫자의 역수(inverse)를 의미 합니다. 다시 설명하면, Document Frequency (DF)는 해당 단어가 출현한 문서의 수가 되고, inverse가 붙어 역수를 취한 것 입니다.

$$\text{TF-IDF}(w, d) = \frac{\text{TF}(w, d)}{\text{DF}(w)}$$

TF는 단어가 문서 내에서 출현한 횟수입니다. 따라서 그 숫자가 클수록 문서 내에서 중요한 단어일 확률이 높습니다. 하지만, 'the'와 같은 단어도 TF값이 매우 클 것입니다. 하지만 'the'가 중요한 경우는 거의 없을 것 입니다. 따라서 이때 IDF가 필요 합니다. DF는 그 단어가 출현한 문서의 숫자를 의미 하기 때문에, 그 값이 클수록 'the'와 같이 일반적으로 많이 쓰이는 단어일 가능성이 높습니다. 따라서 IDF를 구해 TF에 곱해줌으로써, 'the'와 같은 단어들에 대한 penalty를 줍니다. 최종적으로 우리가 얻는 숫자는, 다른 문서들에서는 잘 나타나지 않지만 특정 문서에서만 잘 나타난 경우에 횟수가 높아지기 때문에, 특정 문서에서 얼마나 중요한 역할을 차지하는지 나타내는 수치가 될 수 있습니다.

TF-IDF Example

우리는 아래와 같이 (이미 간단하게 tokenize 된) 여러 문서가 주어졌을 때, 단어들의 TF-IDF를 구하는 방법을 살펴보겠습니다.

번호	내용
1	지능 지수 라는 말 들 어 보 셨 을 겁니다 . 여러분 의 지성 을 일컫 는 말 이 죠 . 그런데 심리 지수 란 건 뭘까요 ? 사람 들 이 특정 한 식 으로 행동 하 는
8	이유 에 대해 여러분 은 얼마나

번호	내용
2	최상의 제시 유형은 학습자에 좌우되는 것이 아니라 학습해야 할 내용에 따라 좌우됩니다. 예를 들어 여러분이 운전하기를 배울 때 실제로 몸으로 체감하는 경험 없이
9	

번호	내용
3	그러나 이 이야기는 세 가지 이유로 인해 신화입니다. 첫째, 가장 중요한 건 실험실 가운은 흰색이 아니라 회색이었다라는 점이죠. 둘째, 참여자들은 실험하기 전에

번호 내용

```
def get_term_frequency(document, word_dict=None):
    if word_dict is None:
        word_dict = {}
    words = document.split()

    for w in words:
        word_dict[w] = 1 + (0 if word_dict.get(w) is None else word_dict[w])

    return word_dict

def get_document_frequency(documents):
    dicts = []
    vocab = set([])
    df = {}

    for d in documents:
        tf = get_term_frequency(d)
        dicts += [tf]
        vocab = vocab | set(tf.keys())

    for v in list(vocab):
        df[v] = 0
        for dict_d in dicts:
            if dict_d.get(v) is not None:
                df[v] += 1

    return df

doc1 = '''
                    .                    .              ?
'''

doc2 = '''
                                                    .
'''
```

```
doc3 = '''
                          .    ,                                    .    ,
'''

def get_tfidf(docs, top_k=30):
    vocab = {}
    tfs = []
    for d in docs:
        vocab = get_term_frequency(d, vocab)
        tfs += [get_term_frequency(d)]
    df = get_document_frequency(docs)

    from operator import itemgetter
    import numpy as np
    sorted_vocab = sorted(vocab.items(), key=itemgetter(1), reverse=True)

    stats = []
    for v, freq in sorted_vocab:
        tfidfs = []
        for idx in range(len(docs)):
            if tfs[idx].get(v) is not None:
                tfidfs += [tfs[idx][v] * np.log(len(docs) / df[v])]
            else:
                tfidfs += [0]

        stats += [(v, freq, tfidfs, max(tfidfs))]

    sorted_tfidfs = sorted(stats, key=itemgetter(3), reverse=True)[:top_k]
    for v, freq, tfidfs, max_tfidfs in sorted_tfidfs:
        print('%s\t%d\t%s' % (v, freq, '\t'.join(['%.4f' % tfidfs[i] for i in rang

>>> get_tfidf([doc1, doc2, doc3])
```

위의 코드를 차례대로 실행한 결과는 아래와 같습니다. 첫번째 문서에서 가장 중요한 단어는 '남자'임을 알 수 있고, 마찬가지로 두번째 문서는 '요인'인 것을 알 수 있습니다.

단어(w)	총 출현 횟수	TF-IDF(d_1)	TF-IDF(d_2)	TF-IDF(d_3)
남자	9	9.8875	0.0000	0.0000

단어(w)	총 출현 횟수	TF-IDF(d_1)	TF-IDF(d_2)	TF-IDF(d_3)
요인	6	0.0000	6.5917	0.0000
심리학	5	5.4931	0.0000	0.0000
멀리	4	4.3944	0.0000	0.0000
시험	4	0.0000	4.3944	0.0000
환경	4	0.0000	4.3944	0.0000
성	4	0.0000	4.3944	0.0000
었	4	0.0000	0.0000	4.3944
제	4	0.0000	0.0000	4.3944
대해	3	3.2958	0.0000	0.0000
나	3	3.2958	0.0000	0.0000
간	3	3.2958	0.0000	0.0000
유형	3	0.0000	3.2958	0.0000
됩니다	3	0.0000	3.2958	0.0000
을까요	3	0.0000	3.2958	0.0000
인증	3	0.0000	3.2958	0.0000
탓	3	0.0000	3.2958	0.0000
유전자	3	0.0000	3.2958	0.0000
유전	3	0.0000	3.2958	0.0000
쌍둥이	3	0.0000	3.2958	0.0000
경우	3	0.0000	3.2958	0.0000
공유	3	0.0000	3.2958	0.0000
참여	3	0.0000	0.0000	3.2958
몸짓	3	0.0000	0.0000	3.2958
거짓말	3	0.0000	0.0000	3.2958
에서	8	2.4328	0.8109	0.0000
지수	2	2.1972	0.0000	0.0000
행동	2	2.1972	0.0000	0.0000
나요	2	2.1972	0.0000	0.0000
또	2	2.1972	0.0000	0.0000

Using Other Features for Similarity

이번엔 다른 방식으로 단어의 유사도를 구하는 방법에 접근 해보겠습니다. 자체적으로 단어에 대한 특성(feature)들을 모아 feature vector로 만들거나

유사도(similarity)를 계산하는 연산을 통해 단어간의 유사도를 구하는 방법입니다. 지금이야 어렵지않게 단어를 vector 형태로 embedding 할 수 있지만, 딥러닝 이전의 시대에는 쉽지 않은 일이었습니다. 이번 섹션을 통해서 딥러닝 이전의 전통적인 방식의 단어간의 유사도를 구하는 방법에 대해 알아보고, 이 방법의 단점과 한계에 대해서 살펴보겠습니다.

Collecting Features

먼저, feature vector를 구성하는 방법들에 대해 살펴보고자 합니다. 결국 아래의 방법들이 하고자 하는 일은 같은 조건에 대해서 비슷한 수치를 반환하는 단어는 비슷한 유사도를 갖도록 벡터를 구성하는 것이라고 할 수 있습니다.

Term-Frequency Matrix

앞서 우리는 TF-IDF에 대해서 살펴 보았습니다. TF-IDF에서 사용되었던 TF (term frequency)는 훌륭한 피쳐(feature)가 될 수 있습니다. 예를 들어 어떤 단어가 각 문서별로 출현한 횟수가 차원별로 구성되면, 하나의 feature vector를 이룰 수 있습니다. 물론 각 문서별 TF-IDF 자체를 사용하는 것도 좋습니다.

```
def get_tf(docs):
    vocab = {}
    tfs = []
    for d in docs:
        vocab = get_term_frequency(d, vocab)
        tfs += [get_term_frequency(d)]

    from operator import itemgetter
    import numpy as np
    sorted_vocab = sorted(vocab.items(), key=itemgetter(1), reverse=True)

    stats = []
    for v, freq in sorted_vocab:
        tf_v = []
        for idx in range(len(docs)):
            if tfs[idx].get(v) is not None:
                tf_v += [tfs[idx][v]]
```

```
            else:
                tf_v += [0]

        print('%s\t%d\t%s' % (v, freq, '\t'.join(['%d' % tf for tf in tf_v])))
>>> get_tf([doc1, doc2, doc3])
```

위의 코드를 사용하여 단어들의 각 문서별 출현횟수를 나타내면 아래와 같습니다.

단어(w)	TF(w)	TF(w, d_1)	TF(w, d_2)	TF(w, d_3)
는	47	15	14	18
을	39	8	10	21
이	32	8	8	16
은	15	6	2	7
가	14	1	7	6
여러분	12	5	6	1
말	11	5	1	5
남자	9	9	0	0
여자	7	5	0	2
차이	7	5	2	0
요인	6	0	6	0
겁니다	5	2	1	2
얼마나	5	4	1	0
심리학	5	5	0	0
학습	5	0	4	1
이야기	5	0	1	4
결과	5	0	4	1
실제로	4	2	1	1
능력	4	3	1	0
멀리	4	4	0	0
시험	4	0	4	0
환경	4	0	4	0
사람	3	1	0	2
동일	3	2	1	0
유형	3	0	3	0
인증	3	0	3	0
유전자	3	0	3	0
수행	3	0	2	1

단어(w)	TF(w)	TF(w, d_1)	TF(w, d_2)	TF(w, d_3)
연구	3	0	2	1
유전	3	0	3	0
쌍둥이	3	0	3	0
경우	3	0	3	0
모두	3	0	1	2
공유	3	0	3	0
인지	3	0	1	2
참여	3	0	0	3
몸짓	3	0	0	3
거짓말	3	0	0	3

여기서 마지막 3개 column이 각 단어별 문서에 대한 출현횟수를 활용한 feature vector가 될 것 입니다. 지금은 문서가 3개밖에 없기 때문에, 사실 정확한 feature vector를 구성했다고 하기엔 무리가 있습니다. 따라서 문서가 많다면 우리는 지금보다 더 나은 feature vector를 구할 수 있을 것 입니다.

하지만 문서가 너무나도 많을 경우에는 벡터의 차원이 너무 커져버릴 수 있습니다. 예를 들어 문서가 10,000개가 있다고 하면 단어 당 10,000차원의 벡터가 만들어질 것 입니다. 문제는 이 10,000차원의 벡터 대부분은 값이 없이 0으로 채워져 있을 것 입니다. 이렇게 벡터의 극히 일부분에만 의미있는 값들로 채워져 있는 벡터를 sparse vector라고 합니다. Sparse vector의 각 차원들은 사실 대부분의 경우 0일 것이기 때문에, 어떤 유의미한 통계를 얻는게 큰 장애물이 될 수 있습니다. 이처럼 희소성 문제는 자연어처리의 고질적인 문제로 작용 합니다.

또한, 단순히 문서에서의 출현 횟수를 가지고 feature vector를 구성하였기 때문에, 많은 정보가 유실되었고, 굉장히 단순화되어 여전히 매우 정확한 feature vector를 구성하였다고 하기엔 무리가 있습니다.

Based on Context Window (Co-occurrence)

함께 나타나는 단어들을 활용한 방법 입니다. 의미가 비슷한 단어라면 쓰임새가 비슷할 것 입니다. 또한, 쓰임새가 비슷하기 때문에, 비슷한 문장 안에서 비슷한 역할로 사용될 것이고, 따라서 함께 나타나는 단어들이 유사할 것 입니다. 이러한 관점에서 우리는 함께 나타나는 단어들이 유사한 단어들의 유사도를 높게 주도록 만들어 줄 것 입니다.

함께 나타나는 단어들을 조사하기 위해서, 우리는 Context Window를 사용하여 windowing을 실행 합니다. (windowing이란 window를 움직이며 window안에 있는 유닛들의 정보를 취합하는 방법을 이릅니다.) 각 단어별로 window 내에 속해 있는 이웃 단어들을 counting하여 matrix로 나타내는 것 입니다.

이 방법은 좀 전에 다룬 문서 내의 단어 출현 횟수(term frequency)를 가지고 feature vector를 구성한 방식보다 좀 더 정확하다고 할 수 있습니다. 하지만, window의 크기라는 하나의 hyper-parameter가 추가되어, 사용자가 그 크기를 정해주어야 합니다. 만약 window의 크기가 너무 크다면, 현재 단어와 너무 관계가 없는 단어들까지 counting 될 수 있습니다. 하지만, 너무 작은 window 크기를 갖는다면, 관계가 있는 단어들이 counting되지 않을 수 있습니다. 따라서, 적절한 window 크기를 정하는 것이 중요 합니다. 또한, window를 문장을 벗어나서도 적용 시킬 것인지도 중요합니다. 문제에 따라 다르지만 대부분의 경우에는 window를 문장 내에만 적용합니다.

python 코드를 통해 아래와 같은 문장들에 대해서 우리는 windowing을 수행 할 수 있습니다.

번호	내용
1	왜
	냐고요
	?
2	산소
	의
	낭비
	였
	지요
	.
3	어느
	날
	,
	저
	는
	요요
	를
	샀
	습니다
	.

번호	내용
4	저 는 회사 의 가치 에 따른 가격 책정 을 돕 습니다 .
5	하지만 내게 매우 내부 적 인 문제 가 생겼 다 .
…	…

번호	내용
9995	고독은 여러분 스스로 찾을 수 있는 곳에 있어서 다른 사람들에게도 다가갈 수 있습니다.

번호	내용
9996	두 번 째 로 이 발견 은 새로운 치료 방법 의 아주 분명 한 행로 를 제시 합니다 . 여기 서부터 무엇 을 해야 하 는지 는 로켓 과학자 가 아니 더라도 알 수 있 잖아요 .

번호	내용
9997	전쟁 전 에 는 시리아 도시 에서 그런 요구 들 이 완전히 무시 되 었 습니다 .

번호	내용
9998	세로로 된 아찔한 암석벽에 둘러쌓여 있으며 숲에 숨겨진 은빛 폭포도 있죠.

번호	내용
9999	얼마간 시간 이 지나 면 큰 소리 는 더 이상 큰 소리 가 아니 게 될 겁니다 .
10000	이러 한 마을 차원 의 아이디어 는 정말 훌륭 한 아이디어 입니다 .

```
def read(fn):
    lines = []

    f = open(fn, 'r')
```

```
    for line in f:
        if line.strip() != '':
            lines += [line.strip()]
    f.close()

    return lines

def get_context_counts(lines, w_size=3):
    co_dict = {}
    for line in lines:
        words = line.split()

        for i, w in enumerate(words):
            for c in words[i - w_size:i + w_size]:
                if w != c:
                    co_dict[(w, c)] = 1 + (0 if co_dict.get((w, c)) is None else

    return co_dict

from operator import itemgetter

fn = 'test.txt'
min_cnt, max_cnt = 0, 100000

lines = read(fn)
co_dict = get_context_counts(lines)
tfs = get_term_frequency(' '.join(lines))
sorted_tfs = sorted(tfs.items(), key=itemgetter(1), reverse=True)

context_matrix = []
row_heads = []
col_heads = [w for w, f in sorted_tfs if f >= min_cnt and f<= max_cnt]
for w, f in sorted_tfs:
    row = []
    if f >= min_cnt and f <= max_cnt:
        row_heads += [w]
        for w_, f_ in sorted_tfs:
            if f_ >= min_cnt and f_ <= max_cnt:
                if co_dict.get((w, w_)) is not None:
```

```
                row += [co_dict[(w, w_)]]
            else:
                row += [0]
        context_matrix += [row]

import pandas as pd

p = pd.DataFrame(data=context_matrix, index=row_heads, columns=col_heads)
```

그리고 이 코드를 통해 얻은 결과의 일부는 아래와 같습니다. 이 결과에 따르면 1000개의 문장(문서)에서는 '습니다'의 context window 내에 마침표가 3616번 등장합니다.

	는	이	을	은	하	의	에	,	들	있	...	다다를	유리병	엉터리	켠	기념관	외침	스프를	이야	금성	악상어
는	0	4227	3554	320	13174	1324	6800	4049	315	8655	...	0	1	0	1	0	0	0	0	2	3
이	8466	0	706	3342	494	3450	1423	2797	12935	4470	...	1	0	0	0	0	1	1	0	1	0
을	5496	3031	0	3064	2249	5075	1875	1119	6720	1783	...	0	1	2	1	2	2	0	1	1	0
은	3298	2764	394	0	40	1994	442	2514	9052	59	...	0	2	2	0	0	0	1	0	3	0
하	13364	2407	9382	717	0	112	1945	700	232	336	...	0	0	0	0	0	0	0	0	0	0
의	3289	1965	295	2386	29	0	409	1858	3699	21	...	0	1	1	0	0	6	3	1	2	3
에	5965	2431	625	1574	391	2945	0	1935	1299	3145	...	7	7	0	5	3	1	1	0	0	2
,	2053	2858	969	1541	2899	675	1305	0	1297	1932	...	0	1	1	1	2	0	0	1	2	0
들	3958	13284	6696	10531	23	5851	1447	2311	0	16	...	0	0	3	0	0	1	0	4	0	0
있	8674	4684	2692	169	2997	20	3802	71	932	0	...	3	0	0	0	0	0	0	0	0	0
습니다	465	3532	1406	218	621	1	663	470	167	11875	...	0	0	0	0	0	0	0	0	0	3
가	2788	1517	351	2008	365	1791	1236	2067	211	2880	...	0	0	1	0	1	0	0	0	0	1
를	2274	2219	96	1768	1247	3336	780	1044	641	179	...	0	0	0	0	0	0	0	0	0	3
고	456	3332	5220	250	8667	43	828	6303	291	12122	...	0	0	0	0	0	1	0	0	0	0
것	10318	7502	5762	5080	3916	414	1296	334	1884	2445	...	3	0	0	0	0	0	0	0	0	0

Figure 6: context windowing을 수행한 결과

앞쪽 출현빈도가 많은 단어들은 대부분 값이 잘 채워져 있는 것을 볼 수 있습니다. 하지만 뒤쪽 출현빈도가 낮은 단어들은 많은 부분이 0으로 채워져 있는 것을 볼 수 있습니다. 출현빈도가 낮은 단어들의 row로 갈 경우에는 그 문제가 더욱 심각해집니다.

위의 context windowing 실행 결과 얻은 feature vector들을 tSNE로 시각화 한 모습은 아래와 같습니다. 딱히 비슷한 단어끼리 모이지 않은 것도 많지만, 운좋게 비슷한 단어끼리 붙어 있는 경우도 종종 볼 수 있습니다.

베네수엘라	1	0	0	0	0	3	1	3	0	0	...	0	0	0	0	0	0	0	0	0	0
표하	0	0	2	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
쿠데타	0	0	0	1	0	1	0	0	1	0	...	0	0	0	0	0	0	0	0	0	0
소용없	1	2	0	1	0	0	1	0	0	0	...	0	0	0	0	0	0	0	0	0	0
유실	0	3	1	2	0	2	0	1	2	0	...	0	0	0	0	0	0	0	0	0	0
거장	0	2	0	1	0	5	0	1	2	0	...	0	0	0	0	0	0	0	0	0	0
물어봅니다	1	2	0	0	0	0	0	0	1	0	...	0	0	0	0	0	0	0	0	0	0
자연스레	2	0	3	0	1	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
대포	0	0	0	0	0	1	1	0	1	0	...	0	0	0	0	0	0	0	0	0	0
분당	4	0	0	0	1	0	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0

Figure 7: 대부분의 값이 0으로 채워진 sparse vector

Get Similarity between Feature Vectors

그럼 이렇게 구해진 feature vector를 어떻게 사용할 수 있을까요? Feature vector는 단어 사이의 유사도를 구할 때 아주 유용하게 쓸 수 있습니다. 그럼 벡터 사이의 유사도 또는 거리는 어떻게 구할 수 있을까요? 아래에서는 두 벡터가 주어졌을 때, 벡터 사이의 유사도 또는 거리를 구하는 방법들에 대해 다루어 보겠습니다. 그리고 PyTorch를 사용하여 해당 수식들을 직접 구현 해 보겠습니다.

```
import torch
```

Manhattan Distance (L1 distance)

$$\mathrm{d}_{\mathrm{L1}}(w, v) = \sum_{i=1}^{d} |w_i - v_i|, \text{ where } w, v \in \mathbb{R}^d.$$

L1 norm을 시용한 Manhattan distance (맨하튼 거리) 입니다. 이 방법은 두 벡터의 각 차원별 값의 차이의 절대값을 모두 합한 값 입니다.

```
def get_l1_distance(x1, x2):
    return ((x1 - x2).abs()).sum()**.5
```

위의 코드는 torch tensor x_1, x_2를 입력으로 받아 L1 distance를 리턴해 주는 코드입니다.

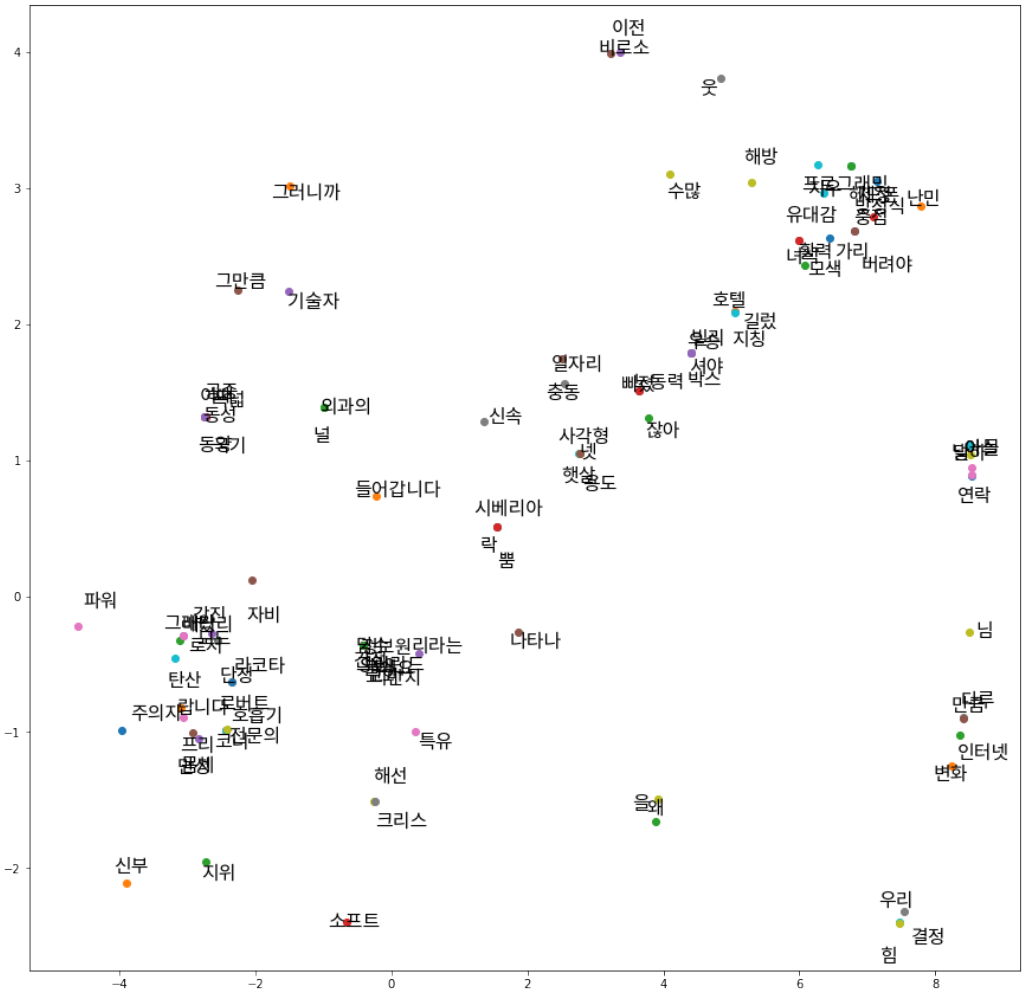


Figure 8: feature vector들을 tSNE로 표현한 모습

Euclidean Distance (L2 distance)

$$\mathrm{d}_{\mathrm{L2}}(w, v) = \sqrt{\sum_{i=1}^{d} (w_i - v_i)^2}, \text{ where } w, v \in \mathbb{R}^d.$$

우리가 가장 친숙한 거리 방법 중의 하나인 Euclidean distance (유클리드 거리)입니다. 각 차원별 값 차이의 제곱의 합에 루트를 취한 형태 입니다.

```
def get_l2_distance(x1, x2):
    return ((x1 - x2)**2).sum()**.5
```

L2 distance를 구하기 위해 torch tensor들읍 입력으로 받아 계산하는 함수의 코드는 위와 같습니다.

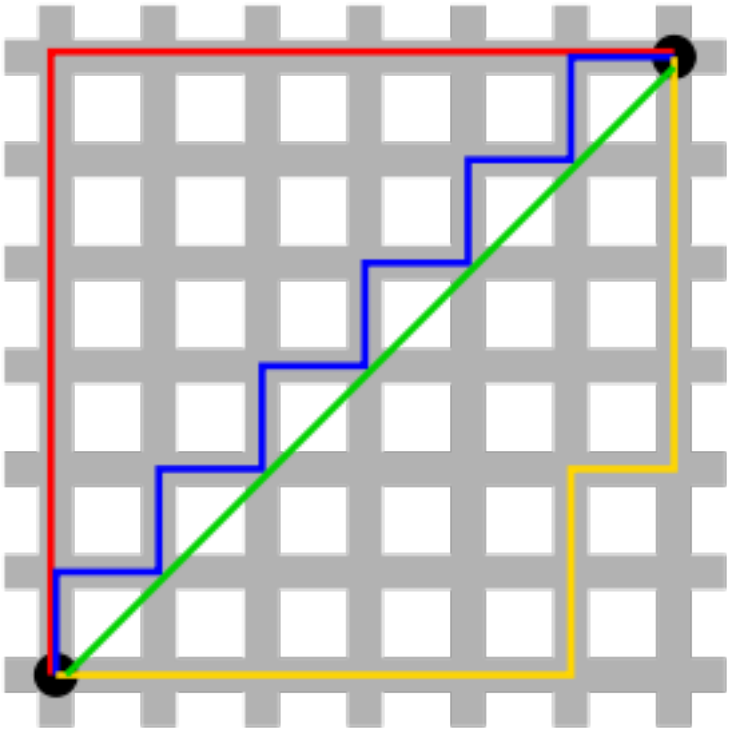


Figure 9: L1 vs L2(초록색) from wikipedia

위의 그림은 Manhanttan distance와 Euclidean distance의 차이를 쉽게 나타낸 그림입니다. Euclidean distance를 의미하는 초록색 선을 제외하고 나머지 Manhattan

distance를 나타내는 세가지 선의 길이는 모두 같습니다. Manhattan distance는 그 이름답게 마치 잘 계획된 도시의 길을 지나가는 듯한 선의 형태를 나타내고 있습니다.

Using Infinity Norm

$$d_\infty(w, v) = \max(|w_1 - v_1|, |w_2 - v_2|, \cdots, |w_d - v_d|), \text{ where } w, v \in \mathbb{R}^d$$

L_1, L_2 distance가 있다면 L_∞ distance도 있습니다. 재미있게도 infinity norm을 이용한 distance는 각 차원별 값의 차이 중 가장 큰 값을 나타냅니다. 위의 수식을 torch tensor에 대해서 계산하는 코드는 아래와 같습니다.

```
def get_infinity_distance(x1, x2):
    return ((x1 - x2).abs()).max()
```

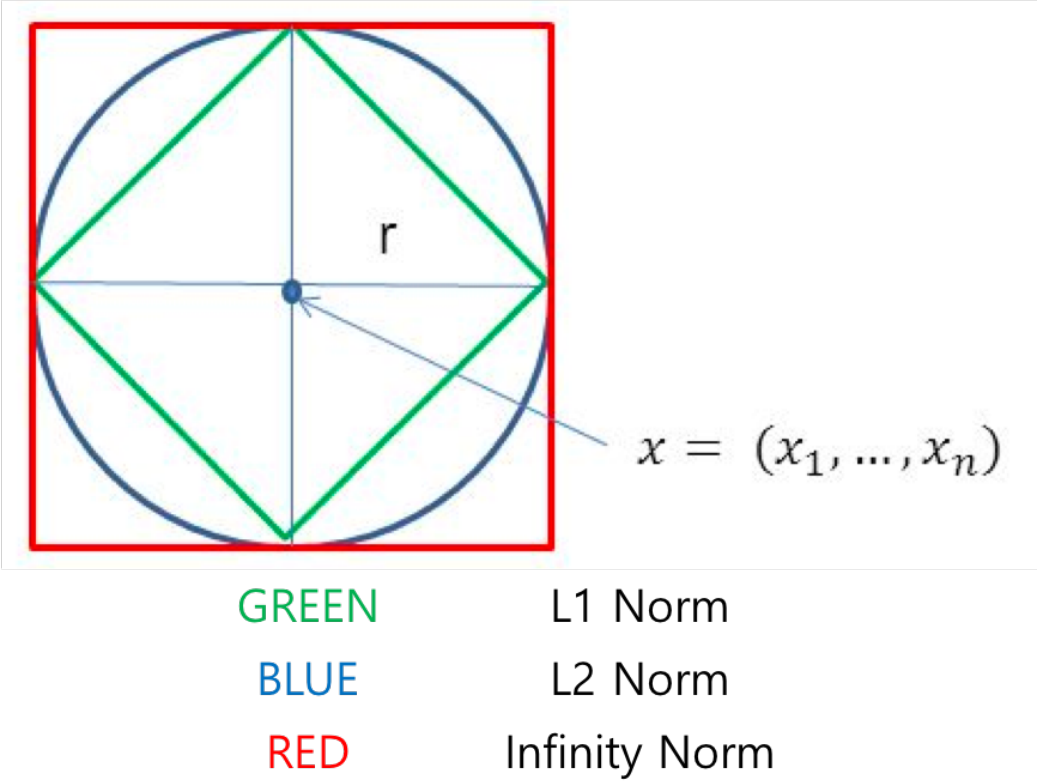


Figure 10: 같은 값 r 크기를 갖는 L_1, L_2, L_∞ 거리를 그림으로 나타낸 모습

위의 그림은 각 L_1, L_2, L_∞ 별로 거리의 크기가 r일때 모습 입니다.

Cosine Similarity

$$\begin{aligned}\text{sim}_{\cos}(w, v) &= \frac{\overbrace{w \cdot v}^{\text{dot product}}}{|w||v|} = \overbrace{\frac{w}{|w|}}^{\text{unit vector}} \cdot \frac{v}{|v|} \\ &= \frac{\sum_{i=1}^{d} w_i v_i}{\sqrt{\sum_{i=1}^{d} w_i^2}\sqrt{\sum_{i=1}^{d} v_i^2}} \\ &\text{where } w, v \in \mathbb{R}^d\end{aligned}$$

위와 같은 수식을 갖는 cosine similarity(코사인 유사도)함수는 두 벡터 사이의 방향과 크기를 모두 고려하는 방법 입니다. 수식에서 분수의 윗변은 두 벡터 사이의 element-wise 곱을 사용하므로 벡터의 내적과 같습니다. 따라서 cosine similarity의 결과가 1에 가까울수록 방향은 일치하고, 0에 가까울수록 수직(orthogonal)이며, -1에 가까울수록 반대 방향임을 의미 합니다. 위와 같이 cosine similarity는 크기와 방향 모두를 고려하기 때문에, 자연어처리에서 가장 널리 쓰이는 유사도 측정 방법 입니다. 하지만 수식 내 윗변의 벡터 내적 연산이나 밑변 각 벡터의 크기(L2 norm)를 구하는 연산이 비싼 편에 속합니다. 따라서 vector 차원의 크기가 클수록 연산량이 부담이 됩니다.

Cosine similarity는 아래와 같이 PyTorch 코드로 나타낼 수 있습니다.

```
def get_cosine_similarity(x1, x2):
    return (x1 * x2).sum() / ((x1**2).sum()**.5 * (x2**2).sum()**.5)
```

Jaccard Similarity

$$\begin{aligned}\text{sim}_{\text{jaccard}}(w, v) &= \frac{|w \cap v|}{|w \cup v|} \\ &= \frac{|w \cap v|}{|w| + |v| - |w \cap v|} \\ &\approx \frac{\sum_{i=1}^{d} \min(w_i, v_i)}{\sum_{i=1}^{d} \max(w_i, v_i)} \\ &\text{where } w, v \in \mathbb{R}^d.\end{aligned}$$

Jaccard similarity는 두 집합 간의 유사도를 구하는 방법 입니다. 수식의 윗변에는 두 집합의 교집합의 크기가 있고, 이를 밑변에서 두 집합의 합집합의 크기로 나누어

줍니다. 이때, Feature vector의 각 차원이 집합의 element가 될 것 입니다. 다만, 각 차원에서의 값이 0 또는 0이 아닌 값이 아니라, 수치 자체에 대해서 Jaccard similarity를 구하고자 할 때에는, 두번째 줄의 수식과 같이 두 벡터의 각 차원의 숫자에 대해서 min, max 연산을 통해서 계산 할 수 있습니다. 이를 PyTorch 코드로 나타내면 아래와 같습니다.

```
def get_jaccard_similarity(x1, x2):
    return torch.stack([x1, x2]).min(dim=0)[0].sum() / torch.stack([x1, x2]).max(
```

Appendix: Similarity between Documents

방금은 단어에 대한 feature를 수집하고 유사도를 구하였다면, 마찬가지로 문서에 대해 feature를 추출하여 문서간의 유사도를 구할 수 있습니다. 예를 들어 문서내의 단어들에 대해 출현 빈도(term frequency)나 TF-IDF를 구하여 vector를 구성하고, 이를 활용하여 vector 사이의 유사도를 구할 수도 있을 것 입니다. # Word Sense Disambiguation

이번 섹션에서는 단어의 유사도가 아닌, 모호성에 대해 다루어 보겠습니다. 하나의 단어는 여러가지 의미를 지닐 수 있습니다. 비슷한 의미의 동형이의어(homonym)인 경우에는 비교적 그 문제가 크지 않을 수 있습니다. 조금은 어색한 문장 표현이나 이해가 될 수 있겟지만, 큰 틀에서는 벗어나지 않기 때문입니다. 하지만 다의어(polysemy)의 경우에는 문제가 커집니다. 앞서 예를 들었던 '차'의 경우에 'tea'의 의미로 해석되느냐, 'car'의 의미로 해석되느냐에 따라 문장의 의미가 매우 달라질것이기 때문입니다. 따라서 이와 같이 단어가 주어졌을 때, 그 의미의 모호성을 없애고 해석하는 것이 매우 중요합니다. 이를 단어 중의성 해소(word sense disambiguation)이라고 합니다.

차를
마시러
공원에
가던
차
안에서
나는
그녀에게
원문 차였다.

G* I
was
kick-
ing
her
in
the
car
that
went
to
the
park
for
tea.

차를
마시러
공원에
가던
차
안에서
나는
그녀에게
원문 차였다.

M* I
was
a
car
to
her,
in
the
car
I
had
a
car
and
went
to
the
park.

차를
마시러
공원에
가던
차
안에서
나는
그녀에게
원문 차였다.

N* I
got
dumped
by
her
on
the
way
to
the
park
for
tea.

차를
마시러
공원에
가던
차
안에서
나는
그녀에게
원문 차였다.

K* I
was
in
the
car
go-
ing
to
the
park
for
tea
and
I
was
in
her
car.

	차를
	마시러
	공원에
	가던
	차
	안에서
	나는
	그녀에게
원문	차였다.
S*	I
	got
	dumped
	by
	her
	in
	the
	car
	that
	was
	go–
	ing
	to
	the
	park
	for
	a
	cup
	of
	tea.

위와 같이 다양한 '차'가 문장에 나타났을 때, 실제 기계번역에 있어서 심각한 성능저하의 원인이 되기도 합니다.

Thesaurus Based Method: Lesk Algorithm

Lesk 알고리즘은 가장 간단한 사전 기반 중의성 해소 방법입니다. 주어진 문장에서 특정 단어에 대해서 의미를 명확히 하고자 할 때 사용 할 수 있습니다. 이를 위해서 Lesk 알고리즘은 간단한 가정을 하나 만듭니다. 문장 내에 같이 등장하는 단어(context)들은 공통 토픽을 공유한다는 것 입니다.

이 가정을 바탕으로 동작하는 Lesk 알고리즘의 개요는 다음과 같습니다. 먼저, 중의성을 해소하고자 하는 단어에 대해서 사전(주로 WordNet)의 의미별 설명과 주어진 문장 내에 등장한 단어의 사전에서 의미별 설명 사이의 유사도를 구합니다. 유사도를 구하는 방법은 여러가지가 있을테지만, 가장 간단한 방법으로는 겹치는 단어의 갯수를 구하는 것이 될 수 있습니다. 이후, 문장 내 단어들의 의미별 설명과 가장 유사도가 높은 (또는 겹치는 단어가 많은) 의미가 선택 됩니다.

예를 들어 NLTK의 WordNet에 'bass'를 검색해 보면 아래와 같습니다.

```
>>> from nltk.corpus import wordnet as wn
>>> for ss in wn.synsets('bass'):
...     print(ss, ss.definition())
...
Synset('bass.n.01') the lowest part of the musical range
Synset('bass.n.02') the lowest part in polyphonic music
Synset('bass.n.03') an adult male singer with the lowest voice
Synset('sea_bass.n.01') the lean flesh of a saltwater fish of the family Serranida
Synset('freshwater_bass.n.01') any of various North American freshwater fish with
Synset('bass.n.06') the lowest adult male singing voice
Synset('bass.n.07') the member with the lowest range of a family of musical instru
Synset('bass.n.08') nontechnical name for any of numerous edible marine and freshw
Synset('bass.s.01') having or denoting a low vocal or instrumental range
```

Lesk 알고리즘 수행을 위하여 간단하게 랩핑(wrapping)하도록 하겠습니다.

```
def lesk(sentence, word):
    from nltk.wsd import lesk

    best_synset = lesk(sentence.split(), word)
    print(best_synset, best_synset.definition())
```

아래와 같이 주어진 문장에서는 'bass'는 물고기의 의미로 뽑히게 되었습니다.

```
>>> sentence = 'I went fishing last weekend and I got a bass and cooked it'
```

```
>>> word = 'bass'
>>> lesk(sentence, word)
```

```
Synset('sea_bass.n.01') the lean flesh of a saltwater fish of the family Serranida
```

또한, 아래의 문장애서는 음악에서의 역할을 의미하는 것으로 추정되었습니다.

```
>>> sentence = 'I love the music from the speaker which has strong beat and bass'
>>> word = 'bass'
>>> lesk(sentence, word)
```

```
Synset('bass.n.02') the lowest part in polyphonic music
```

여기까지 보면 Lesk 알고리즘은 비교적 잘 동작하는 것으로 보입니다. 하지만, 비교적 정확하게 예측해낸 위의 두 사례와 달리, 아래의 문장에서는 전혀 다른 의미로 예측하는 것을 볼 수 있습니다.

```
>>> sentence = 'I think the bass is more important than guitar'
>>> word = 'bass'
>>> lesk(sentence, word)
```

```
Synset('sea_bass.n.01') the lean flesh of a saltwater fish of the family Serranida
```

이처럼 Lesk 알고리즘은 명확한 장단점을 지니고 있습니다. WordNet과 같이 잘 분류된 사전이 있다면, 쉽고 빠르게 중의성해소를 해결할 수 있을 것 입니다. 또한 WordNet에서 이미 단어별로 몇개의 의미를 갖고 있는지 잘 정의 해 놓았기 때문에, 크게 고민 할 필요도 없습니다. 하지만 보다시피 사전의 단어 및 의미에 대한 설명에 크게 의존하게 되고, 설명이 부실하거나 주어진 문장에 큰 특징이 없을 경우 단어 중의성해소 능력은 크게 떨어지게 됩니다. 그리고 WordNet이 모든 언어에 대해서 존재하는 것이 아니기 때문에, 사전이 존재하지 않는 언어에 대해서는 Lesk 알고리즘 자체의 수행이 어려울 수도 있습니다.

Monty-hall Problem

$$
\begin{aligned}
P(C=2|A=0,B=1) &= \frac{P(A=0,B=1,C=2)}{P(A=0,B=1)} \\
&= \frac{P(B=1|A=0,C=2)P(A=0,C=2)}{P(A=0,B=1)} \\
&= \frac{P(B=1|A=0,C=2)P(A=0)P(C=2)}{P(B=1|A=0)P(A=0)} \\
&= \frac{1 \times \frac{1}{3}}{\frac{1}{2}} = \frac{2}{3},
\end{aligned}
$$

where $P(B=1,A=0) = \frac{1}{2},\ P(C=2) = \frac{1}{3},$ and $P(B=1|A=0,C=2) = 1.$

$$
\begin{aligned}
P(C=0|A=0,B=1) &= \frac{P(A=0,B=1,C=0)}{P(A=0,B=1)} \\
&= \frac{P(B=1|A=0,C=0)P(A=0,C=0)}{P(A=0,B=1)} \\
&= \frac{P(B=1|A=0,C=0)P(A=0)P(C=0)}{P(B=1|A=0)P(A=0)} \\
&= \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{1}{3},
\end{aligned}
$$

$$
\text{where } P(B=1|A=0,C=0) = \frac{1}{2}
$$

Selectional Preference

이번 섹션에서는 selectional preference라는 개념에 대해서 다루어 보도록 하겠습니다. 문장은 여러 단어의 시퀀스로 이루어져 있습니다. 따라서 각 단어들은 문장내 주변의 단어들에 따라 그 의미가 정해지기 마련입니다. Selectional preference는 이를 좀 더 수치화하여 나타내 줍니다. 예를 들어 '마시다'라는 동사에 대한 목적어는''클래스에 속하는 단어가 올 확률이 매우 높습니다. 따라서 우리는 '차'라는 단어가''클래스에 속하는지" 클래스에 속하는지 쉽게 알 수 있습니다. 이런 성질을 이용하여 우리는 단어 중의성 해소(WSD)도 해결 할 수 있으며, 여러가지 문제들(syntactic disambiguation, semantic role labeling)을 수행할 수 있습니다.

Selectional Preference Strength

앞서 언급한 것처럼 selectional preference는 단어와 단어 사이의 관계(예: verb-object)가 좀 더 특별한 경우에 대해 수치화 하여 나타냅니다. 술어(predicate) 동사(예: verb)가 주어졌을 때, 목적어(예: object)관계에 있는 headword 단어(보통은 명사가 될 겁니다.)들의 분포는, 평소 문서 내에 해당 명사(예: object로써 noun)가 나올 분포와 다를 것 입니다. 그 분포의 차이가 크면 클수록 해당 술어(predicate)는 더 강력한 selectional preference를 갖는다고 할 수 있습니다. 이것을 Philip Resnik은 [Resnik et al.1997]에서 Selectional Preference Strength라고 명명하고 KL-divergence를 사용하여 정의하였습니다.

$$\begin{aligned} S_R(w) &= \mathrm{KL}(P(C|w)||P(C)) \\ &= -\sum_{c \in \mathcal{C}} P(c|w) \log \frac{P(c)}{P(c|w)} \\ &= -\mathbb{E}_{C \sim P(C|w)}[\log \frac{P(C)}{P(C|W=w)}] \end{aligned}$$

위의 수식을 해석하면, selectional preference strength $S_R(w)$은 w가 주어졌을 때의 object class C의 분포 $P(C|w)$와 그냥 해당 class들의 prior(사전) 분포 $P(C)$와의 KL-divergence로 정의되어 있음을 알 수 있습니다. 즉, selectional preference strength는 술어(predicate)가 headword로 특정 클래스를 얼마나 선택적으로 선호(selectional preference)하는지에 대한 수치라고 할 수 있습니다.

예를 들어 '' 클래스의 단어는 '' 클래스의 단어보다 나타날 확률이 훨씬 높을 것 입니다. 이때, '사용하다'라는 동사(verb) 술어(predicate)가 주어진다면, 동사-목적어(verb-object) 관계에 있는 headword로써의''클래스의 확률은" 클래스의 확률보다 낮아질 것 입니다.

Selectional Association

이제 그럼 술어와 특정 클래스 사이의 선택 관련도를 어떻게 나타내는지 살펴보겠습니다. Selectional Association, $A_R(w, c)$은 아래와 같이 표현됩니다.

$$A_R(w, c) = -\frac{P(c|w) \log \frac{P(c)}{P(c|w)}}{S_R(w)}$$

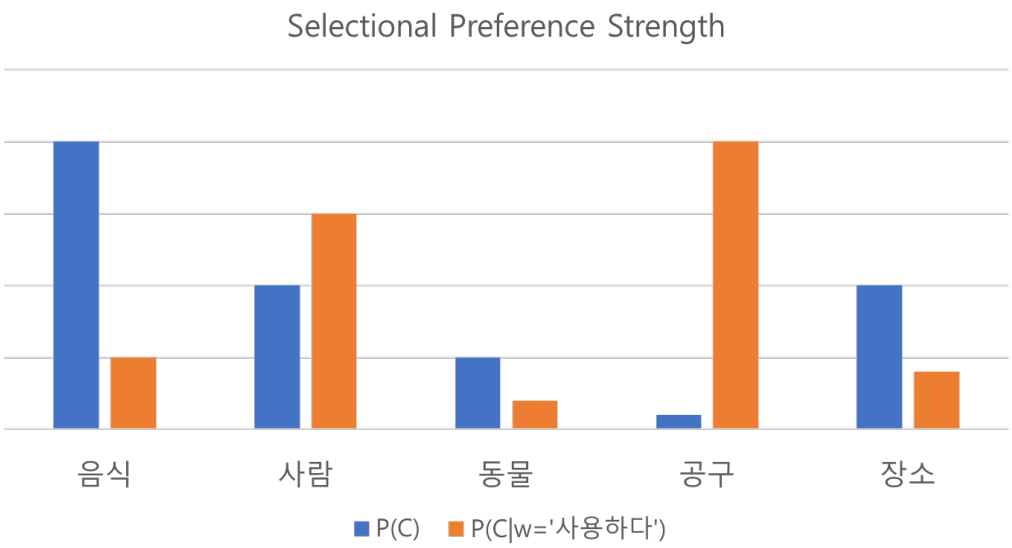


Figure 11: 클래스의 사전 확률 분포와 술어가 주어졌을 때의 확률 분포 변화

위의 수식에 따르면, selectional preference strength가 낮은 술어(predicate)에 대해서 윗변의 값이 클 경우에는 술어와 클래스 사이에 더 큰 selectional association(선택 관련도)를 갖는다고 정의 합니다. 즉, selectional preference strength가 낮아서, 해당 술어(predicate)는 클래스(class)에 대한 선택적 선호 강도가 낮음에도 불구하고, 특정 클래스만 유독 술어에 영향을 받아서 윗변이 커질수록 selectional association의 수치도 커집니다.

예를 들어 어떤 아주 일반적인 동사에 대해서는 대부분의 클래스들이 prior(사전)확률 분포와 비슷하게 여전히 나타날 것입니다. 따라서 selectional preference strength, $S_R(w)$는 0에 가까울것 입니다. 하지만 그 가운데 해당 동사와 붙어서 좀 더 나타나는 클래스의 목적어가 있다면, selectional association $A_R(w, c)$는 매우 높게 나타날 것입니다.

Selectional Preference and WSD

눈치가 빠른 분들은 이미 눈치 채셨겠지만, 우리는 이런 selectional preference의 특성을 이용하여 단어 중의성 해소(Word Sense Disambiguation)에 활용할 수 있습니다. '마시다'라는 동사에 '차'라는 목적어가 함께 있을 때, 우리는 selectional preference를 통해서 '차'는" 클래스에 속한다고 말할 수 있을 것 입니다.

문제는 '차'가''또는" 클래스에 속하는 것을 알아내는 것 입니다. 우리가 가지고 있는 코퍼스는 단어들로 표현되어 있지, 클래스로 표현되어 있지는 않습니다. 만약 우리는 단어가 어떤 클래스들에 속하는지 미리 알고 있다면 단어들의 출현 빈도를 세어 클래스의 확률 분포를 estimation(추정)할 수 있을 것 입니다. 결국 이를 위해서는 사전에 정의되어 있는 지식 또는 데이터셋이 필요할 것 입니다.

Selectional Preference based on WordNet

이때, WordNet이 위력을 발휘합니다. 우리는 WordNet을 통해서 '차(car)'의 hypernym을 알 수 있고, 이것을 클래스로 삼으면 우리가 필요한 정보들을 얻을 수 있습니다. 영어 WordNet에서의 예를 들어 'eat' 뒤에 있는 'bass'의 hypernym을 통해 먹는 물고기 'bass'인지, 악기 'bass'인지 구분 할 수 있을 것 입니다. [Resnik et al.1997]에서는 술어와 클래스 사이의 확률 분포를 정의하기 위한 출현빈도를 계산하는 수식을 아래와 같이 제안하였습니다.

$$\text{Count}_R(w, c) \approx \sum_{h \in c} \frac{\text{Count}_R(w, h)}{|\text{Classes}(h)|}$$

클래스 c에 속하는 headword, h는 실제 코퍼스(corpus)에 나타난 단어로써, 술어(predicate) w와 함께 출현한 headword h의 빈도를 세고, h가 속하는 클래스들의 set의 크기, $|\text{Classes}(h)|$로 나누어 줍니다. 그리고 이를 클래스 c에 속하는 모든 단어(headword)에 대해서 수행한 후, 이를 합한 값은 $Count_R(w, c)$를 근사(approximation)합니다.

$$\hat{c} = \underset{c \in \mathcal{C}}{\text{argmax}} A_R(w, c), \text{ where } \mathcal{C} = \text{hypernym}(h).$$

이를 통해 우리는 predicate w와 headword h가 주어졌을 때, h의 클래스 c를 추정한 $\hat{c}$를 구할 수 있습니다.

Selectional Preference Evaluation using Pseudo Word

단어 중의성 해소(WSD)를 해결할 수 있는 방법 중에 하나로, selectional preference를 살펴 보았습니다. Selectional preference를 잘 해결할 수 있다면 아마도 단어 중의성 해소 문제도 잘 해결 될 것 입니다. 그럼 selectional preference를 어떻게 평가 할 수

있을까요? 정교한 테스트셋을 설계하고 만들어서 selectional preference의 성능을 평가 할 수 있겠지만, 좀 더 쉽고 general한 방법은 없을까요?

Pseudo Word가 하나의 해답이 될 수 있습니다. Pseudo word는 두개의 단어가 인위적으로 합성되어 만들어진 단어를 이릅니다. 실제 일상 생활에서 쓰이기보다는 단순히 두 단어를 합친 것 입니다. 예를 들어 banana와 door를 합쳐 banana-door라는 pseudo word를 만들어 낼 수 있습니다. 이 banana-door라는 단어는 사실 실생활에서 쓰일리 없는 단어입니다. 하지만 우리는 이 단어가 eat 또는 open이라는 동사 술어(verb predicate)와 함께 headword object로써 나타났을때, eat에 대해서는 banana를 선택해야 하고, open에 대해서는 door를 선택하도록 해야 올바른 selectional preference 알고리즘을 만들거나 구현했음을 확인할 수 있습니다.

[Chambers et al.2010]

Similarity-based Selectional Preference

위와 같이 selectional preference를 WordNet등의 도움을 받아 구현하는 방법을 살펴보았습니다. 하지만, WordNet이라는것은 아쉽게도 모든 언어에 존재하지 않으며, 새롭게 생겨난 신조어들도 반영되어 있지 않을 가능성이 매우 높습니다. 따라서 WordNet과 같은 thesaurus에 의존하지 않고 selectional preference를 구할 수 있으면 매우 좋을 것 입니다. [Erk et al.2007]에서는 단어간의 유사도를 통해 thesaurus에 의존하지 않고 간단하게 selectional preference를 구하는 방법을 제시하였습니다.

$$(w, h, R), \text{ where } R \text{ is a relationship, such as verb-object.}$$

이전과 같이 문제 정의는 비슷합니다. 술어(predicate) w와 headword h, 그리고 두 단어 사이의 관계 R이 tuple로 주어집니다. 이때, selectional association, $A_R(w, h_0)$은 아래와 같이 정의 할 수 있습니다.

$$A_R(w, h_0) = \sum_{h \in \text{Seen}_R(w)} \text{sim}(h_0, h) \cdot \phi_R(w, h)$$

이때, weight $\phi_R(w, h)$는 uniform하게 1로 주어도 되고, 아래와 같이 Inverse Document Frequency (IDF)를 사용하여 정의 할 수도 있습니다.

$$\phi_R(w, h) = \text{IDF}(h)$$

또한, sim함수는 이전에 다루었던 cosine similarity나 jaccard similarity를 포함하여 다양한 유사도 함수를 사용할 수 있습니다.

다만, 이전에 다루었던 selectional association은 $A_R(w, c)$을 다루었는데, 지금은 $A_R(w, h_0)$을 다루는 것이 차이점이긴 합니다. 하지만 우리는 좀 전에 selectional preference의 문제를 선호하는 클래스를 알아내는 것이 아닌, pseudo word와 같이 두 개의 단어가 주어졌을 때 선호하는 단어를 고르는 문제로 만들었습니다. 따라서 큰 문제가 되지 않습니다.

이 방법은 쉽게 WordNet과 같은 thesaurus 없이 쉽게 selectional preference를 계산할 수 있게 합니다. 하지만 유사도를 비교하기 위해서 $Seen_R(w)$ 함수를 통해 대상 단어를 선정하기 때문에, 코퍼스에 따라서 유사도를 구할 수 있는 대상이 달라지게 됩니다. 따라서 이에 따랏 커버리지 문제가 발생할 수 있습니다.

Example

```
from konlpy.tag import Mecab

def count_seen_headwords(lines, predicate='VV', headword='NNG'):
    mecab = Mecab()
    seen_dict = {}

    for line in lines:
        pos_result = mecab.pos(line)

        word_h = None
        word_p = None
        for word, pos in pos_result:
            if pos == predicate or pos[:3] == predicate + '+':
                word_p = word
                break
            if pos == headword:
                word_h = word

        if word_h is not None and word_p is not None:
            seen_dict[word_p] = [word_h] + ([] if seen_dict.get(word_p) is None el

    return seen_dict
```

```
def get_selectional_association(predicate, headword, lines, dataframe, metric):
    v1 = torch.FloatTensor(dataframe.loc[headword].values)
    seens = count_seen_headwords(lines)[predicate]

    total = 0
    for seen in seens:
        try:
            v2 = torch.FloatTensor(dataframe.loc[seen].values)
            total += metric(v1, v2)
        except:
            pass

    return total
```

```
def wsd(predicate, headwords):
    selectional_associations = []
    for h in query_h:
        selectional_associations += [get_selectional_association(query_p, h, lines

    print(selectional_associations)
```

```
>>> wsd(' ', [' ', ' ', ' '])
[tensor(6.1853), tensor(3.9723), tensor(3.8503)]
```

Conclusion

이번 챕터에서는 단어의 의미에 대해서 다뤄보고, 의미의 유사성이나 모호성에 대해서 다루어보았습니다. 단어는 겉의 discrete한 형태와 달리 내부적으로는 non-discrete한 '의미(sense)'를 갖고 있습니다. 따라서 우리는 단어의 겉 모양이 다르더라도 의미가 유사할 수 있음을 알고 있습니다. 이렇게 의미가 유사한 단어들간의 유사도를 계산 할 수 있다면, 코퍼스(corpus)로부터 분포나 특징(feature)들을 훈련 할 때 좀 더 많은 정보를 얻고 정확한 훈련을 할 수 있습니다. 예를 들어 꼭 source 단어가 아니더라도 그 source 단어와 유사한 단어들로부터 정보를 얻어올 수 있고, 그 정보를 source 단어와 유사한 만큼만 사용하면 될 것입니다.

따라서, 이러한 자연어처리의 염원 아래, 사람이 직접 한땀한땀 정성들여 만든

WordNet이라는 사전이 등장하게 되었고, WordNet을 활용하여 단어 사이의 유사도(거리)를 계산할 수도 있게 되었습니다. 하지만 이렇게 WordNet과 같은 사전(thesaurus)을 구축하는 것은 너무나도 엄청난 일이므로, 사전이 없이 유사도를 구할 수 있으면 더 좋을 것 입니다.

비록 사전이 없이 코퍼스만 가지고 특징(feature)을 추출하여 단어를 벡터로 만든다면, WordNet의 정교한 지식은 이용할 수 없겠지만, 훨씬 더 간단한 작업이 될 것 입니다. 더욱이 코퍼스의 크기가 커질수록 추출된 특징들은 점점 더 정확해 질 것이고, feature vector는 더욱 정확해 질 것 입니다. Feature vector가 추출 된 이후에는 cosine similarity나 L2 distance등의 metric을 통해서 유사도를 계산할 수 있습니다.

하지만, 이전에 보았듯이, 단어 사전의 크기가 30,000~50,000이 넘는 현실에서 이렇게 추출된 feature vector의 차원은 단어 사전의 크기와 맞먹습니다. 단어 대신 문서를 feature로 사용하더라도 주어진 문서의 숫자만큼 vector의 차원이 만들어질 것 입니다. vector가 큰 문제는 차치하고서라도, 더 큰 문제는 그 차원의 대부분의 값들이 0으로 채워져 있다는 것 입니다. 즉, 각 차원에 숫자가 나타나는 경우는 0이 나타나는 경우에 비해서 현저히 적습니다. 따라서 이런 0으로 가득한 sparse vector를 통해 무엇인가 배우고자 할 때에 큰 장애로 작용합니다.

이러한 sparsity(희소성)문제는 자연어처리의 가장 큰 특징 입니다. 단어는 discrete한 심볼로 이루어져 있기 때문입니다. 따라서 전통적인 자연어처리에서는 이러한 희소성 문제로 인해서 정말 큰 어려움을 겪었습니다. 사실 많은 분들이 잘 알고 있듯이, 요즘 딥러닝에서는 단어의 feature vector를 이런 sparse vector로 만들기보단 embedding 기법을 통해 dense vector로 만들어 사용 합니다. 이런 차이점이 바로 딥러닝에서의 자연어처리가 전통적인 방식의 자연어처리에 비해서 성능이 월등한 이유 입니다. 이제 우리는 앞으로 다룰 챕터에서 딥러닝을 통해 이런 자연어처리의 문제를 잘 해결하는 방법들을 소개해보고자 합니다.