자연어 처리(NLP)

자연어처리란?

[자연어 처리의 정의]

- **자연어(Nautral Language)**: 인간이 일상에서 사용하는 언어
- **자연어 처리(Natural Laguage Processing)**: 기계가 자연어를 이해하고 해석하여 처리할 수 있도록 하는 일(NLP)
- **자연어 처리(NLP)**와 **텍스트 분석(Text Mining)**은 다른 개념
  - 자연어 처리는 **기계가 인간의 언어를 해석**하는데 중점을 두고 있음
  - 텍스트 분석은 **텍스트에서 의미 있는 정보를 추출하여 인사이트**를 얻는데 더 중점을 두고 있

[NLP 활용 분야]

- **텍스트 분류(Text Classification):** 텍스트가 특정 분류, 카테고리에 속하는 것을 예측하는 기법(예:스펨메일분류)
- **감성 분석(Sentiment Analysis):** 텍스트에 나타나는 감정/기분 등의 주관적 요소를 분석하는 기법(예: 영화 및 제품의 리뷰)
- **텍스트 요약(Summarization):** 중요한 주제를 추출하여 요약하는 기법(예: 토픽 모델링(Topic Modeling))
- **텍스트 군집화(Clustering)와 유사도 측정:** 비슷한 유형의 텍스트에 대해 군집화하는 기법
- **기계 번역(Translation):** 구글 번역기나 파파고와 같은 번역기에도 활용
- **대화 시스템 및 자동 질의 응답 시스템:** 애플의 시리나 삼성 갤럭시의 빅스비, 챗봇 등이 이에 속함

자연어처리란?

[NLP 처리 프로세스]

- **텍스트 전처리(Text Preprocessing):** 텍스트 정규화 작업을 수행
  - 대/소문자 변경, 특수문자 삭제, 이모티콘 삭제 등의 전처리 작업, 단어(Word) 토큰화 작업
  - 불용어(Stop word) 제거 작업
  - 어근 추출(Stemming/Lemmatization)

- **피처 벡터화 (Feature Vectorization):** 전처리된 텍스트에서 피처를 추출하고 여기에 벡터 값을 할당하는 것
  - 대표적인 기법: BOW(Bag of words)와 Word2Vec

- **머신러닝 모델링:** 피처 벡터화된 데이터에 대하여 모델을 수립하고 학습/예측을 하는 단계

자연어처리란?

[NLP를 위한 라이브러리]

- **NLTK(National Language Toolkit for Python)**
  - 파이썬 NLP 패키지의 시조새에 해당
  - 가장 기본적인 패키지이며, 오랫동안 연구되었으며, 최근에는 수행 속도가 다소 느려 제대로 활용 되지 않음
  - 초기 공부시 NLP를 이해하기 좋은 라이브러리임

- **Gensim**
  - 토픽 모델링 분야에서 가장 두각을 보이는 NLP 패키지
  - SpaCy와 함께 가장 많이 사용되는 NLP 패키지

- **SpaCy:** 수행 성능이 좋아 최근 가장 많이 활용되고 있는 NLP 패키지

텍스트 토큰화(Text Tokenization)

- NLP에서 텍스트 자체를 바로 피처로 사용할 수는 없음.
- 따라서 사전에 텍스트 전처리 작업이 반드시 필요함
- 텍스트 전처리를 위해서는 클렌징, 토큰화, 불용어 제거, 정규화 등의 작업이 필요

[말뭉치(Corpus, 코퍼스)]

- **사전적 의미**

  - 말뭉치 또는 코퍼스(Corpus)는 자연언어 연구를 위해 특정한 목적을 가지고 언어의 표본을 추출한 집합이다.
  - 즉, 우리가 사용하는 텍스트 표본으로 생각하면 됨

- **토큰(Token)**

  - 문법적으로 더 이상 나눌 수 없는 언어요소를 뜻함
  - 따라서 텍스트 토큰화(Text Tokenization)란 말뭉치로부터 토큰을 분리하는 작업을 뜻함

    예) 말뭉치: "There is an apple" => 토큰화: "There", "is", "an", "apple"
  - 문장 토큰화: 텍스트에서 문장을 분리하는 작업
  - 단어 토큰화: 문자아에서 단어를 토큰으로 분리하는 작업

텍스트 토큰화(Text Tokenization)

[문장 토큰화(Sentence Tokenization)]

- 문장의 마침표(.), 개행문자(₩n), 느낌표(!), 물음표(?) 등 문장의 마지막을 뜻하는 기호에 따라 분리하는 것
- NLTK를 이용한 문장 토큰화(한국어의 경우 Konlpy 이용)

```
from nltk import sent_tokenize
text_sample = 'The Matrix is everywhere its all around us, here even in this room. You can see
tokenized_sentences = sent_tokenize(text_sample)
print(tokenized_sentences)
```

```
['The Matrix is everywhere its all around us, here even in this room.', 'You can see it out yo
```

- 3개의 문장으로 분리

텍스트 토큰화(Text Tokenization)

[문장 토큰화(Sentence Tokenization)]

- NLTK를 이용한 문장 토큰화(Ph.D를 포함하는 문장)

```
text_sample = 'I am actively looking for Ph.D. students. and you are a Ph.D student.'
tokenized_sentences = sent_tokenize(text_sample)
print(tokenized_sentences)
```

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

- 마침표가 있지만 정확하게 분리

텍스트 토큰화(Text Tokenization)

[단어 토큰화 (Word Tokenization)]

- 단어 토큰화는 기본적으로 띄어쓰기를 기준으로 구분
- 영어는 보통 띄어쓰기로 토큰이 구분
- 한국어는 띄어쓰기 만으로 토큰을 구분하기 어려움, 심지어 띄어쓰기가 잘못 되어 있는 경우도 허다함

```
from nltk import word_tokenize

sentence = "The Matrix is everywhere its all around us, here even in this room."
words = word_tokenize(sentence)
print(words)
```

```
['The', 'Matrix', 'is', 'everywhere', 'its', 'all', 'around', 'us', ',', 'here', 'even', 'in',
```

- 띄어쓰기를 기반으로 분리를 하되 콤마(,)와 마침표(.)는 별도의 토큰으로 구분

텍스트 토큰화(Text Tokenization)

[단어 토큰화 (Word Tokenization)]

- 어퍼스트로피(')가 있는 경우

```
from nltk.tokenize import WordPunctTokenizer
sentence = "Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as ch
words = WordPunctTokenizer().tokenize(sentence)
print(words)
```

```
['Don', "'", 't', 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr', '.', 'Jo
```

- Don't는 Don, ', t로 Jone's는 Jone, ', s로 구분

텍스트 토큰화(Text Tokenization)

[단어 토큰화 (Word Tokenization)]

- keras의 text_to_word_sequence을 이용한 경우

```
from tensorflow.keras.preprocessing.text import text_to_word_sequence
sentence = "Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as ch
words = text_to_word_sequence(sentence)
print(words)
```

```
["don't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', 'mr', "jone's", 'orphanage'
```

- 모든 알파벳을 소문자로 바꾸고, 구두점(컴마, 마침표 등)을 없애고, 어퍼스트로피(')도 보존하여 토큰을 제대로 구분

불용어(Stop word) 제거

[불용어(Stop word)]

- **분석에 큰 의미가 없는 단어**
  - 예) the, a, an, is, I, my 등과 같이 문장을 구성하는 필수 요소지만 문맥적으로 큰 의미가 없는 단어
- **따라서 사전에 제거가 필요**

[불용어 확인]

```
import nltk
nltk.download('stopwords')

print('영어 불용어 갯수:',len(nltk.corpus.stopwords.words('english')))
print(nltk.corpus.stopwords.words('english')[:40])
```

```
영어 불용어 갯수: 179
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you
```

불용어(Stop word) 제거

[불용어 제거]

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

example = "Family is not an important thing. It's everything."
stop_words = set(stopwords.words('english'))

word_tokens = word_tokenize(example)

result = []
for token in word_tokens:
    if token not in stop_words:
        result.append(token)

print(word_tokens)
print(result)
```

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

- **is, not, an과 같은 불용어가 제외됨**

불용어(Stop word) 제거

[한국어 불용어]

- **nltk의 stopwords에서는 한국어 불용어를 지원하지 않음**
- **일반적으로 사용하는 한국어 불용어 리스트: https://www.ranks.nl/stopwords/korean**
  - 한국어 불용어를 처리하는 가장 좋은 방법은 txt나 csv 파일에 불용어를 직접 정리해놓고, 이를 불러와서 사용하는 것이 정확성을 높일 수 있음

```
# 사용자 stop_words
stop_words=['를', '및', '나', '점', '알', '이', '룸', '자', '안', '외', '것', '수', '단', '쭈' '투', '파이썬', '책', '출력

wt_cls_word=[wtcl for wtcl in wt_text if wtcl not in stop_words]
#print(wt_cls_word)
```

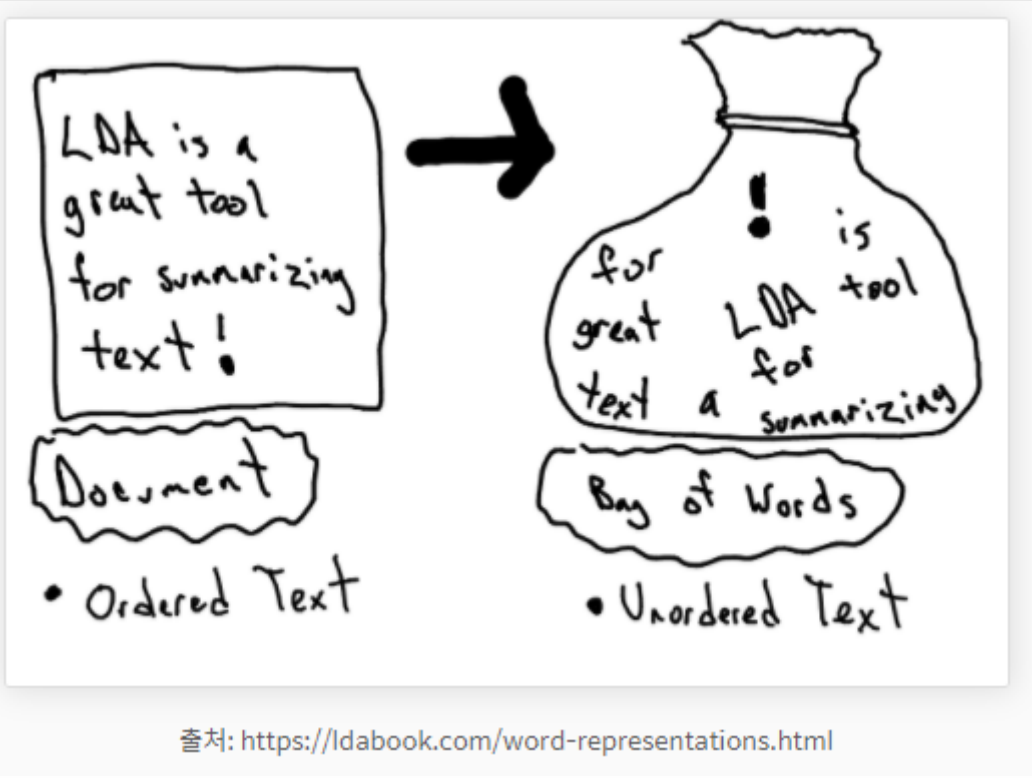
- **re 모듈(정규식 표현)을 이용하기도 하나 정확한 의미의 불용어 제거는 아님**

  예) re.sub('[^a-zA-Zㄱ-ㅣ가-힣0-9?!#]',"",doc)

Bag of Words (BOW)

[Bag of Words모델]

- 단어들의 문맥이나 순서를 무시하고, 단어들에 대해 빈도 값(frequency)을 부여해 피쳐 값을 만드는 모델
- Bag of Words == 단어들의 가방
- 문서 내 모든 단어를 한꺼번에 가방(Bag) 안에 넣은 뒤에 흔들어서 섞는다는 의미



출처: https://ldabook.com/word-representations.html

Bag of Words (BOW)

[BOW를 기반으로 피처 추출하기]

- **문장1: 'My wife likes to watch baseball games and my daughter likes to watch baseball games too'**
- **문장2: 'My wife likes to play baseball'**

1) 문장1과 문장 2에 있는 모든 단어에서 중복을 제거하고 각 단어를 칼럼 형태로 나열하고
각 단어에 고유의 인덱스를 아래와 같이 부여

'and':0, 'baseball':1, 'daughter':2, 'games':3, 'likes':4, 'my':5, 'play':6, 'to':7, 'too':8, 'watch':9, 'wife':10

2) 개별 문장에서 해당 단어가 나타나는 횟수(frequency)를 각 단어(단어 인덱스)에 기재
예) baseball은 문장 1,2에서 총 2번 나타나며, play는 문장 2에서만 1번 나타남

index	0	1	2	3	4	5	6	7	8	9	10
	and	baseball	daughter	games	likes	my	play	to	too	watch	wife
문장1	1	2	1	2	2	2	0	2	1	2	1
문장2	0	1	0	0	1	1	1	1	0	0	1

Bag of Words (BOW)

[Bag of Words(BOW)의 단점]

- **문맥 의미(Semantic Context) 반영 부족**

  - BOW는 단어의 순서를 고려하지 않기 때문에 문맥적인 의미가 무시됨

  - 이를 보완하기 위해 n_gram기법을 활용할 수는 있지만, 제한적임

- **희소 행렬 문제**

  - BOW로 피처 벡터화하면 희소 행렬 형태의 데이터가 만들

  - 문장이 굉장히 많은 문서에서 BOW 피처 벡터화를 하면 칼럼이 매우 많아짐

  - 희소 행렬(Sparse Matrix): 상당히 많은 칼럼으로 구성되어 있는 행렬에서 대부분의 값이 0으로 채워진 행렬

    * 반대로 대부분의 값이 0이 아닌 값으로 채워진 행렬을 밀집 행렬(Dense Matrix)이라고 함

Bag of Words (BOW)

[BOW 피처 벡터화 (BOW Feature Vectorization)]

- **머신러닝 모델은 문자로 된 피처를 바로 사용할 수 없음, 따라서 BOW를 활용해 문자 숫자화 해야함**
- **이 처럼 텍스트를 특정 의미가 있는 숫자형 값인 벡터 값으로 변환해야 하는데, 이를 피처 벡터화(Feature Vectorization)라고 함**
- **BOW 모델에서 피처 벡터화란 모든 문서에서 모든 단어를 칼럼 형태로 나열하고 해당 단어에 빈도 값을 부여하는 것을 뜻**
- **BOW의 피처 벡터화 방식**
  - 카운트 기반 벡터화(CountVectorizer)
  - TF-IDF(Term Frequency - Inverse Document Frequency) 기반 벡터화

BOW 피처 벡터화

[카운트 기반 벡터화(CountVectorizer)]

- 값이 높을수록 중요한 단어로 인식
- 사이킷런의 CountVectorizer 클래스를 활용하여 적용

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = ['you know I want your love. because I love you.']
vector = CountVectorizer()
print(vector.fit_transform(corpus).toarray()) # 코퍼스로부터 각 단어의 빈도 수를 기록한다.
print(vector.vocabulary_) # 각 단어의 인덱스가 어떻게 부여되었는지를 보여준다.
```

```
[[1 1 2 1 2 1]]
{'you': 4, 'know': 1, 'want': 3, 'your': 5, 'love': 2, 'because': 0}
```

- CountVectorizer는 기본적으로 2사이 이상의 문자에 대해서만 토큰으로 인식

BOW 피처 벡터화

[불용어(Stop words)를 제거한 BOW]

```
from sklearn.feature_extraction.text import CountVectorizer

text=["Family is not an important thing. It's everything."]
vect = CountVectorizer(stop_words=["the", "a", "an", "is", "not"])
print(vect.fit_transform(text).toarray())
print(vect.vocabulary_)
```

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

- **the, a, an, is, not을 불용어로 정의했기 때문에, 출력된 토큰에 the, a, an, is, not 이 제거됨**

BOW 피처 벡터화

[불용어(Stop words)를 제거한 BOW]

- **CountVectorizer에서 제공하는 불용어를 이용한 제거**

```
from sklearn.feature_extraction.text import CountVectorizer

text=["Family is not an important thing. It's everything."]
vect = CountVectorizer(stop_words="english")
print(vect.fit_transform(text).toarray())
print(vect.vocabulary_)
```

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

BOW 피처 벡터화

[TF-IDF(Term Frequency - Inverse Document Frequency)]

- 개별 문서에서 자주 등장하는 단어에 높은 가중치를 주되, 모든 문서에서 전반적으로 자주 등장하는 단어에 대해서는 패널티를페널티를 주는 방식으로 값을 부여
- 모든 문서에서 자주 등장하는 단어에는 페널티를 주고, 해당 문서에서만 자주 등장하는 단어에 높은 가중치를 주는 방식
- 그렇게 함으로써 해당 단어가 실질적으로 중요한 단어인지 검사하는 것
- 문서의 양이 많을 경우에는 일반적으로 카운트 기반의 벡터화보다 TF-IDF 방식의 벡터화를 사용

BOW 피처 벡터화

[TF-IDF(Term Frequency - Inverse Document Frequency)]

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = [
    'you know I want your love',
    'I like you',
    'what should I do ',
]
vector = CountVectorizer()
print(vector.fit_transform(corpus).toarray()) #
print(vector.vocabulary_) # 각 단어의 인덱스가
```

```
[[0 1 0 1 0 1 0 1 1]
 [0 0 1 0 0 0 0 1 0]
 [1 0 0 0 1 0 1 0 0]]
{'you': 7, 'know': 1, 'want': 5, 'your': 8, 'lov
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [
    'you know I want your love',
    'I like you',
    'what should I do ',
]
tfidfv = TfidfVectorizer().fit(corpus)
print(tfidfv.transform(corpus).toarray())
print(tfidfv.vocabulary_)
```

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

희소 행렬 (Sparse Matrix)

[희소 행렬 (Sparse Matrix)]

- CountVectorizer, TfidfVectorizer를 이용해 피처 벡터화를 하면 상당히 많은 칼럼이 생성됨
  모든 문서에 포함된 모든 고유 단어를 피처로 만들어주기 때문이며 수만개에서 수십만개의 단어가 만들어짐
- 이렇게 대규모의 행렬이 생기면 각 문서에 포함된 단어의 수는 제한적이기 때문에 행렬의 대부분의 값은 0으로 채워짐
  이렇게 대부분 ix)이라고 함

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

출처: Quora

- 반대로 대부분의 값이 0이 아닌 값으로 채워진 행렬은 밀집 행렬(Dense Matrix)이라고 함

희소 행렬 (Sparse Matrix)

[희소 행렬 표현 방식]

- **COO형식**

  0이 아닌값 [1, 5, 1, 4, 3, 2, 5, 6, 3, 2, 7, 8, 1]

  행 위치 배열 = [0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

  열 위치 배열 = [2, 5, 0, 1, 3, 4, 5, 1, 3, 0, 3, 5, 0]

0	0	1	0	0	5
1	4	0	3	2	5
0	6	0	3	0	0
2	0	0	0	0	0
0	0	0	7	0	8
1	0	0	0	0	0

- **CSR 형식**
  - 행 위치 배열 = [0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

    ==> 시작 위치 배열인 [0, 2, 7, 9, 10, 12]로 변환
  - 총항목이 13개이므로 최종 배열은 [0, 2, 7, 9, 10, 12, 13]
- **사이킷런의 CountVectorizer, TfidfVectorizer 클래스로 피처 벡터화된 행렬은 모두 CSR 형식**

희소 행렬 (Sparse Matrix)

[희소 행렬 표현 방식]

- **COO형식**

  0이 아닌값 [1, 5, 1, 4, 3, 2, 5, 6, 3, 2, 7, 8, 1]

  행 위치 배열 = [0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

  열 위치 배열 = [2, 5, 0, 1, 3, 4, 5, 1, 3, 0, 3, 5, 0]

0	0	1	0	0	5
1	4	0	3	2	5
0	6	0	3	0	0
2	0	0	0	0	0
0	0	0	7	0	8
1	0	0	0	0	0

- **CSR 형식**
  - 행 위치 배열 = [0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

    ==> 시작 위치 배열인 [0, 2, 7, 9, 10, 12]로 변환
  - 총항목이 13개이므로 최종 배열은 [0, 2, 7, 9, 10, 12, 13]
- **사이킷런의 CountVectorizer, TfidfVectorizer 클래스로 피처 벡터화된 행렬은 모두 CSR 형식**

코사인 유사도(Cosine Similarity)

[유사도란?]

- **문서 유사도란 말그대로 문서와 문서 간의 유사도가 어느정도인지 나타내는 척도**
- **문서 간 유사도를 측정해 지금 보고 있는 뉴스와 가장 유사한 뉴스를 추천, 줄거리를 기반으로 내가 본 영화와 가장 유사한 영화를 추천 등 가능**

[코사인 유사도 (Cosine Similarity)]

- **벡터와 벡터 간의 유사도를 비교할 때 두 벡터 간의 사잇각을 구해서 얼마나 유사한지 수치로 나타낸 것**
  - 벡터 방향이 비슷할수록 두 벡터는 서로 유사
  - 벡터 방향이 90도 일때는 두 벡터 간의 관련성이 없음
  - 벡터 방향이 반대가 될수록 두 벡터는 반대 관계

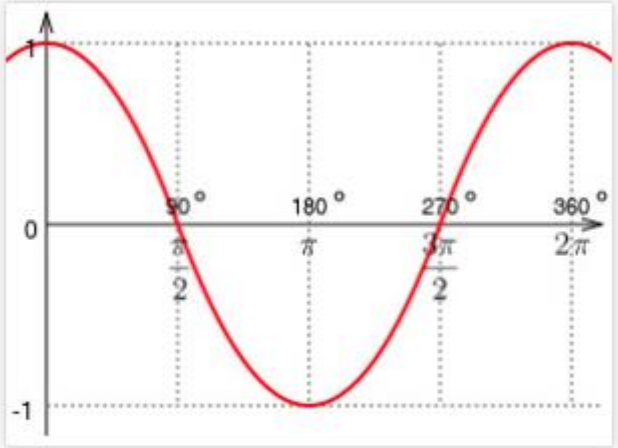
출처: 딥 러닝을 이용한 자연어 처리 입문

코사인 유사도(Cosine Similarity)

[코사인 유사도 (Cosine Similarity)]

- **두 벡터 간의 사잇각을 코사인 씌워준 값을 통해 구할 수 있음(중학교 코사인 함수?)**

- θ를 두 벡터 간의 사잇각이라고 했을 때 θ=0이면, cosθ = 1
  * 코사인 유사도가 1이면 두 벡터는 완전히 동일한 벡터
- 두 벡터 간의 사잇각이 90도이면, 코사인 유사도가 0이되고 두 벡터는 상관 관계가 없다고 말함
- 두 벡터 간의 사잇각이 180도 이면 코사인 유사도는 -1이며 두 벡터는 완전히 반대인 벡터



출처: calculat.org

* 피처 벡터 행렬은 음수값이 없으므로 코사인 유사도가 음수가 되지 않음, 따라서 코사인 유사도는 0~1 사이의 값을 갖음

$$A \cdot B = |A| * |B| * \cos\Theta$$

slideshare.net 벡터와 매트릭스

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

코사인 유사도(Cosine Similarity)

[코사인 유사도 (Cosine Similarity) 구하기]

```
import numpy as np

def cos_similarity(v1, v2):
    dot_product = np.dot(v1, v2)
    l2_norm = (np.sqrt(sum(np.square(v1))) * np.sqrt(sum(np.square(v2))))
    similarity = dot_product / l2_norm

    return similarity
```

```
from sklearn.feature_extraction.text import TfidfVectorizer

doc_list = ['if you take the blue pill, the story ends' ,
            'if you take the red pill, you stay in Wonderland',
            'if you take the red pill, I show you how deep the rabbit hole goes']

tfidf_vect_simple = TfidfVectorizer()
feature_vect_simple = tfidf_vect_simple.fit_transform(doc_list)

print(feature_vect_simple.shape)
print(type(feature_vect_simple))
```

```
(3, 18)
<class 'scipy.sparse.csr.csr_matrix'>
```

문서가 3개, 총 단어의 갯수(중복 제거)가 18개

TF-IDF 벡터화 한 행렬은 3 X 18 행렬

코사인 유사도(Cosine Similarity)

[코사인 유사도 (Cosine Similarity) 구하기]

```
# TFidfVectorizer로 transform()한 결과는 Sparse Matrix이므로 Dense Matrix로 변환.
feature_vect_dense = feature_vect_simple.todense()

#첫번째 문장과 두번째 문장의 feature vector  추출
vect1 = np.array(feature_vect_dense[0]).reshape(-1,)
vect2 = np.array(feature_vect_dense[1]).reshape(-1,)

#첫번째 문장과 두번째 문장의 feature vector로 두개 문장의 Cosine 유사도 추출
similarity_simple = cos_similarity(vect1, vect2)
print('문장 1, 문장 2 Cosine 유사도: {0:.3f}'.format(similarity_simple))
```

* feature_vect_simple.todense()는 희소 행렬인 feature_vect_simple을 밀집 행렬로 변환하는 코드

코사인 유사도(Cosine Similarity)

[코사인 유사도 (Cosine Similarity) 구하기]

- **사이킷런은 코사인 유사도를 측정하기 위한 cosine_similarity() API를 제공**
- **cosine_similarity(비교 기준이 되는 문서의 피처 행렬, 비교하고자 하는 문서의 피처 행렬)**

```
# TFidfVectorizer로 transform()한 결과는 Sparse Matrix이므로 Dense Matrix로 변환.
feature_vect_dense = feature_vect_simple.todense()

#첫번째 문장과 두번째 문장의 feature vector  추출
vect1 = np.array(feature_vect_dense[0]).reshape(-1,)
vect2 = np.array(feature_vect_dense[1]).reshape(-1,)

#첫번째 문장과 두번째 문장의 feature vector로 두개 문장의 Cosine 유사도 추출
similarity_simple = cos_similarity(vect1, vect2)
print('문장 1, 문장 2 Cosine 유사도: {0:.3f}'.format(similarity_simplefrom
sklearn.metrics.pairwise import cosine_similarity

similarity_simple_pair = cosine_similarity(feature_vect_simple[0] , feature_vect_simple)
print(similarity_simple_pair)
```

```
# 코사인 유사도 행렬로 표현

similarity_simple_pair = cosine_similarity(feature_vect_simple , feature_vect_simple)
print(similarity_simple_pair)
```

워드 임베딩 (Word Embedding)

단어를 밀집 행렬로 표현하는 것을 워드 임베딩(Word Embedding) 혹은 단어의 분산 표현(Distributed Representation)이라고 함

우선 원-핫 인코딩을 해보겠습니다. 원-핫 인코딩의 결과는 희소 행렬입니다.

빨 = (1, 0, 0, 0, 0, 0, 0)
주 = (0, 1, 0, 0, 0, 0, 0)
노 = (0, 0, 1, 0, 0, 0, 0)
초 = (0, 0, 0, 1, 0, 0, 0)
파 = (0, 0, 0, 0, 1, 0, 0)
남 = (0, 0, 0, 0, 0, 1, 0)
보 = (0, 0, 0, 0, 0, 0, 1)

7개의 색을 표현하기 위해 7개의 열이 필요함

빨 = (255, 0, 0)
주 = (255, 50, 0)
노 = (255, 255, 0)
초 = (0, 255, 0)
파 = (0, 0, 255)
남 = (0, 5, 255)
보 = (100, 0, 255)

이와 유사하게 문자를 밀집 벡터로 나타내는 것을 워드 임베딩이라고 함
대표적인 워드 임베딩 방법으로는 **Word2Vec, GloVe, Elmo**가 있음

워드투벡터(Word2Vec)

[Word2Vec]

- **워드 임베딩(Word Embedding) 방법론 중 하나**

1. 한국이라는 단어에서 수도에 해당하는 서울을 빼줍니다.
   한국에서 서울이라는 특성을 뺐으니 나라에 해당하는 껍데기 의미만 남음.
   거기에 파리를 더해주면 프랑스가 됨
   즉, 나라에 해당하는 껍데기에 파리라는 프랑스 수도를 더해주니 그 단어는
   프랑스가 되는 것

한국 - 서울 + 파리 = 프랑스

어머니 - 아버지 + 여자 = 남자

아버지 + 여자 = 어머니

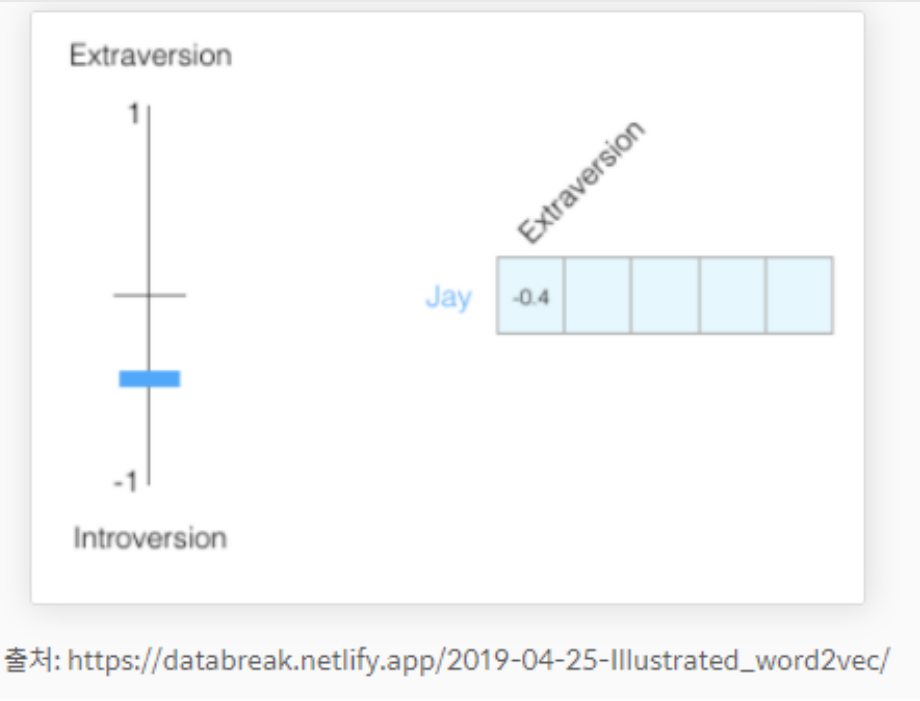
워드투벡터(Word2Vec)

[Word2Vec]

- 예시) 사람의 성격을 벡터로 표현하는 것을

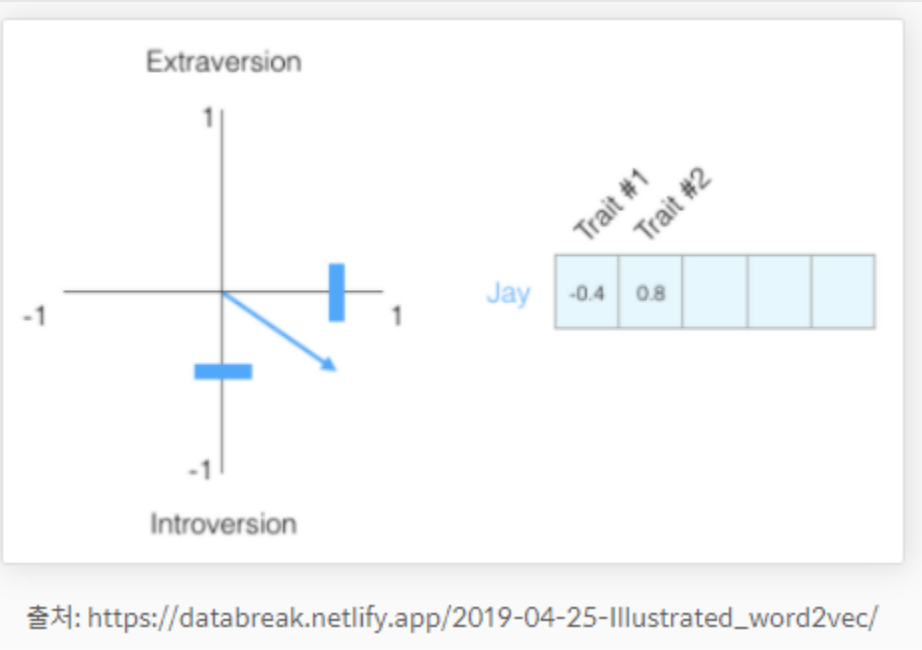
1. 우선 외향성&내향성 정도를 -1에서 1사이의 값으로 표현한다고 가정

    * 1에 가까울수록 외향적인 성격이고 -1에 가까울수록 내향적인 성격



출처: https://databreak.netlify.app/2019-04-25-Illustrated_word2vec/

2. 이성적&감성적 요소를 나타내는 값도 매김



출처: https://databreak.netlify.app/2019-04-25-Illustrated_word2vec/

워드투벡터(Word2Vec)

[Word2Vec]

- 두가지 요소를 고려시하여 세 사람의 외향성&내향성, 이성&감정 정도를 측정

  - 벡터 형식으로 세 사람의 성격이 표현

출처: https://databreak.netlify.app/2019-04-25-Illustrated_word2vec/

  - 세 사람 중 누구의 성격이 서로 비슷한지 확인
    (코사인 유사도를 통해 측정)

출처: https://databreak.netlify.app/2019-04-25-Illustrated_word2vec/

워드투벡터(Word2Vec)

[CBOW(Continuous Bag of Words)와 Skip-gram]

- **CBOW 모델: 맥락으로부터 타깃을 예측하는 용도의 신경망**
- **Skip-gram은 반대로 타깃으로부터 맥락을 예측하는 용도의 신경망**

※ '타깃'은 중앙 단어이고, '맥락'은 그 주변 단어들임

[예시]

You ___ goodbye and I say hello.

여기서 ___에 들어갈 단어를 예측하는 모델이 CBOW임,

- ____에 들어갈 단어가 '타깃(Target)'이고, 타깃의 주변 단어인 'You'와 'goodbye'는 '맥락(Context)'입,
- 따라서 여기서는 ____에 들어갈 단어로 'say'가 가장 적합한 답변일 것임('dog'나 'chair'와 같은 단어는 부적합)

___ say ____ and I say hello.

반대로, 위와 같이 중앙의 단어(타깃)로부터 주변의 여러 단어(맥락)를 예측하는 모델을 Skip-gram이라 함

- 하나의 단어로부터 그 주변 단어들을 예측(매우 어려운 문제임)

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

[Window size]

You say _____ and I say hello.라는 문장이 있을 때, 타깃 단어를 예측하기 위해 사용할 앞뒤 맥락 단어의 갯수를 Window라고 함.

- window가 1이면 say와 and를 통해 타깃 단어를 예측
- window가 2이면 You, say, and, I를 통해 타깃을 예측

중심 단어	주변 단어
The fat cat sat on the mat	[1, 0, 0, 0, 0, 0]	[0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0]
The fat cat sat on the mat	[0, 1, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0]
The fat cat sat on the mat	[0, 0, 1, 0, 0, 0]	[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0]
The fat cat sat on the mat	[0, 0, 0, 1, 0, 0]	[0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1]
The fat cat sat on the mat	[0, 0, 0, 0, 1, 0]	[0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1]
The fat cat sat on the mat	[0, 0, 0, 0, 0, 1]	[0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1]
The fat cat sat on the mat	[0, 0, 0, 0, 0, 1]	[0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1, 0]

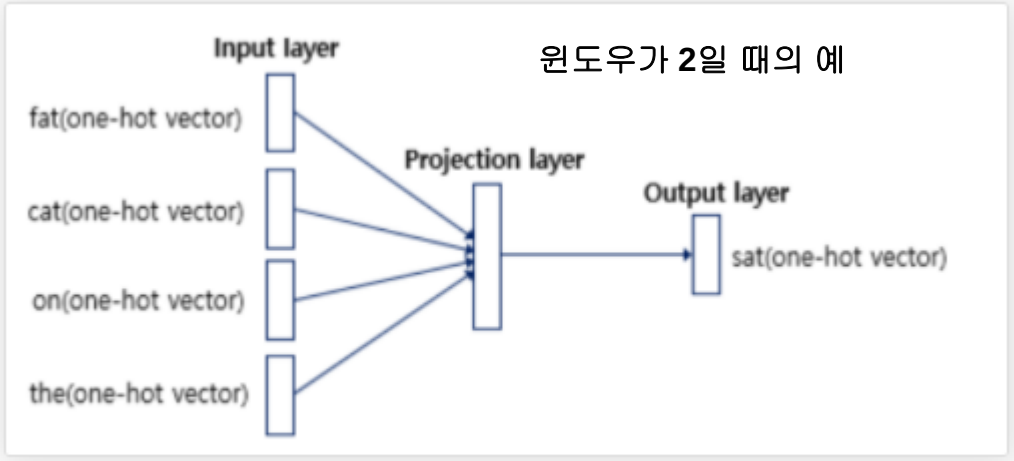
출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

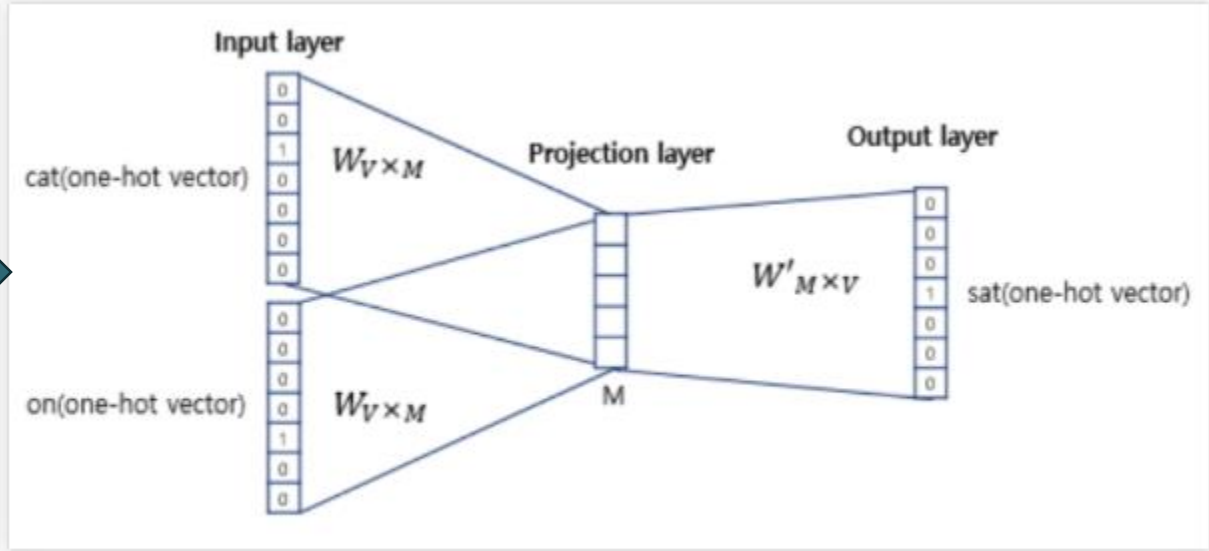
- CBOW 신경망을 학습시켜 나온 최적의 가중치가 바로 워드 임베딩된 벡터임
- 맥락 단어로부터 타깃 단어를 예측하는 모델임

  즉, CBOW 모델의 입력(input)은 맥락 단어의 원-핫 벡터이고, 출력(output)은 타깃 단어의 원-핫 벡터



출처: 딥러닝을 이용한 자연어처리

확대



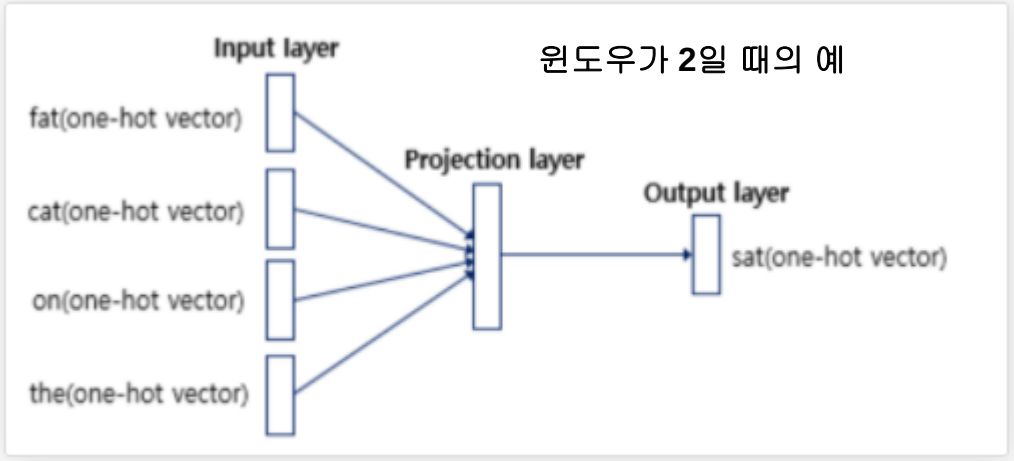
출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

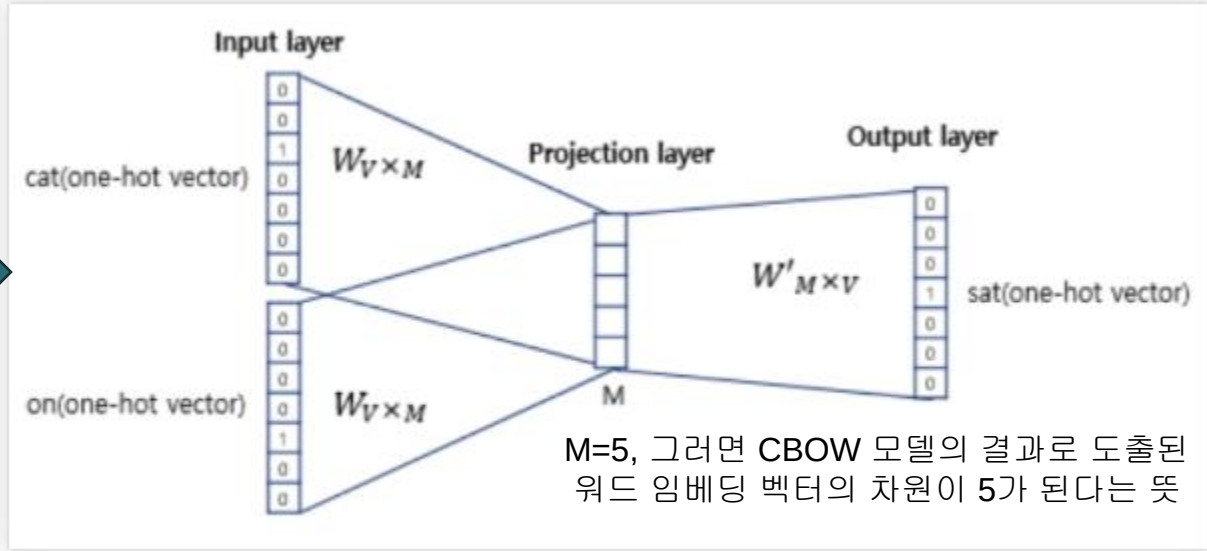
- CBOW 신경망을 학습시켜 나온 최적의 가중치가 바로 워드 임베딩된 벡터임
- 맥락 단어로부터 타깃 단어를 예측하는 모델임

  즉, CBOW 모델의 입력(input)은 맥락 단어의 원-핫 벡터이고, 출력(output)은 타깃 단어의 원-핫 벡터



출처: 딥러닝을 이용한 자연어처리

확대



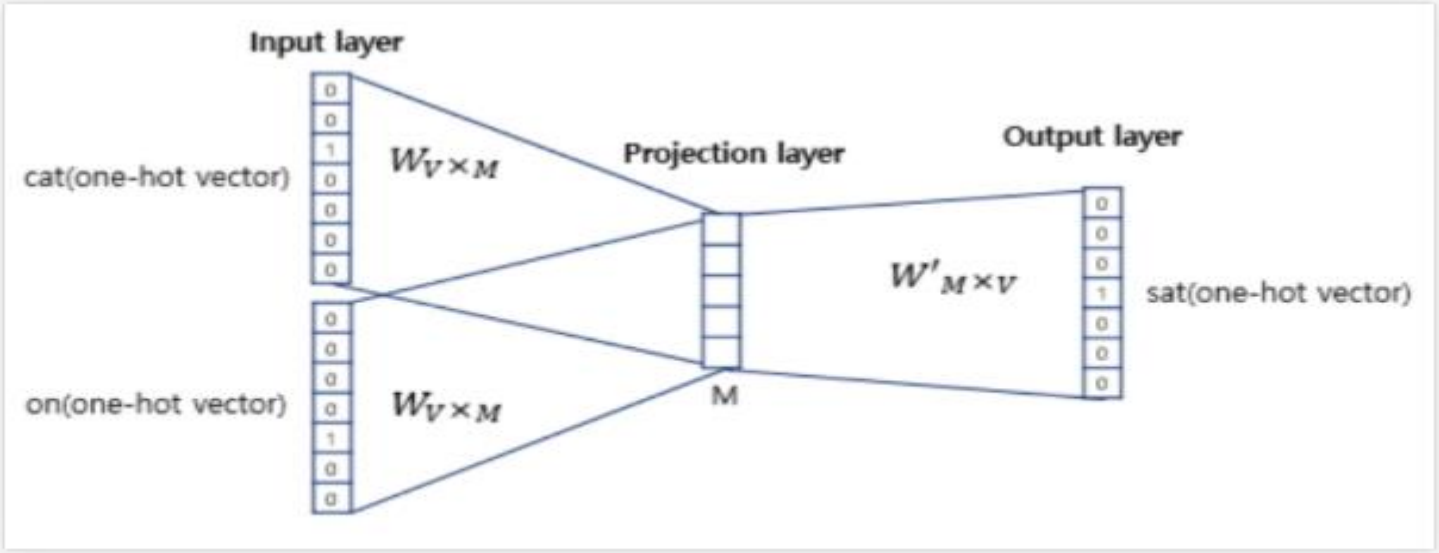
M=5, 그러면 CBOW 모델의 결과로 도출된 워드 임베딩 벡터의 차원이 5가 된다는 뜻

출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

- 입력층(input layer)에서의 가중치 행렬 W의 크기는 V x M
- V는 단어의 갯수이므로 V = 7, 가중치 행렬 W의 크기는 7 x 5 임,
- 반면 출력층의 W' 행렬의 크기는 M x V, 즉 5 x 7 (W와 W'는 서로 다른 행렬임)
- CBOW 신경망 모델은 입력 벡터(맥락 단어)를 통해 출력 벡터(타깃 단어)를 맞추기 위해 계속해서 학습하며 이 가중치 행렬 W와 W'을 갱신함



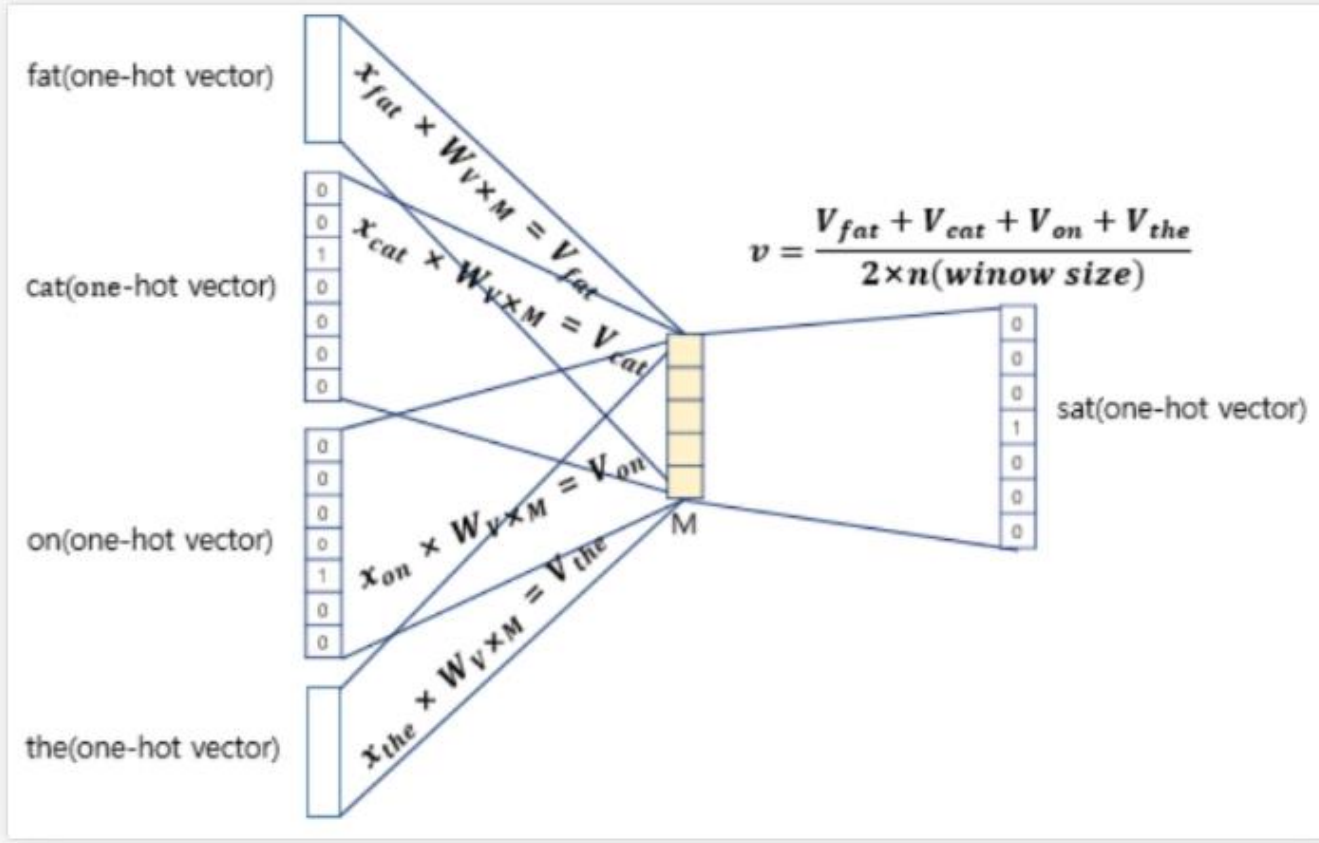
출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

- 가중치 행렬 W가 결국 우리가 구하고자 하는 워드 임베딩 결과, 즉 좋은 워드 임베딩 값을 구하기 위해서는 가중치 행렬 W를 잘 학습해야 함

- 입력인 맥락 단어가 총 4개이므로 각 입력 원-핫 벡터와 가중치 행렬 W를 곱한 벡터 v가 4개 도출
- 구해진 모든 v벡터 (v_fat, v_eat, v_on, v_the)를 더한 뒤 4(2 x (window size))로 나누어 평균 벡터를 구함

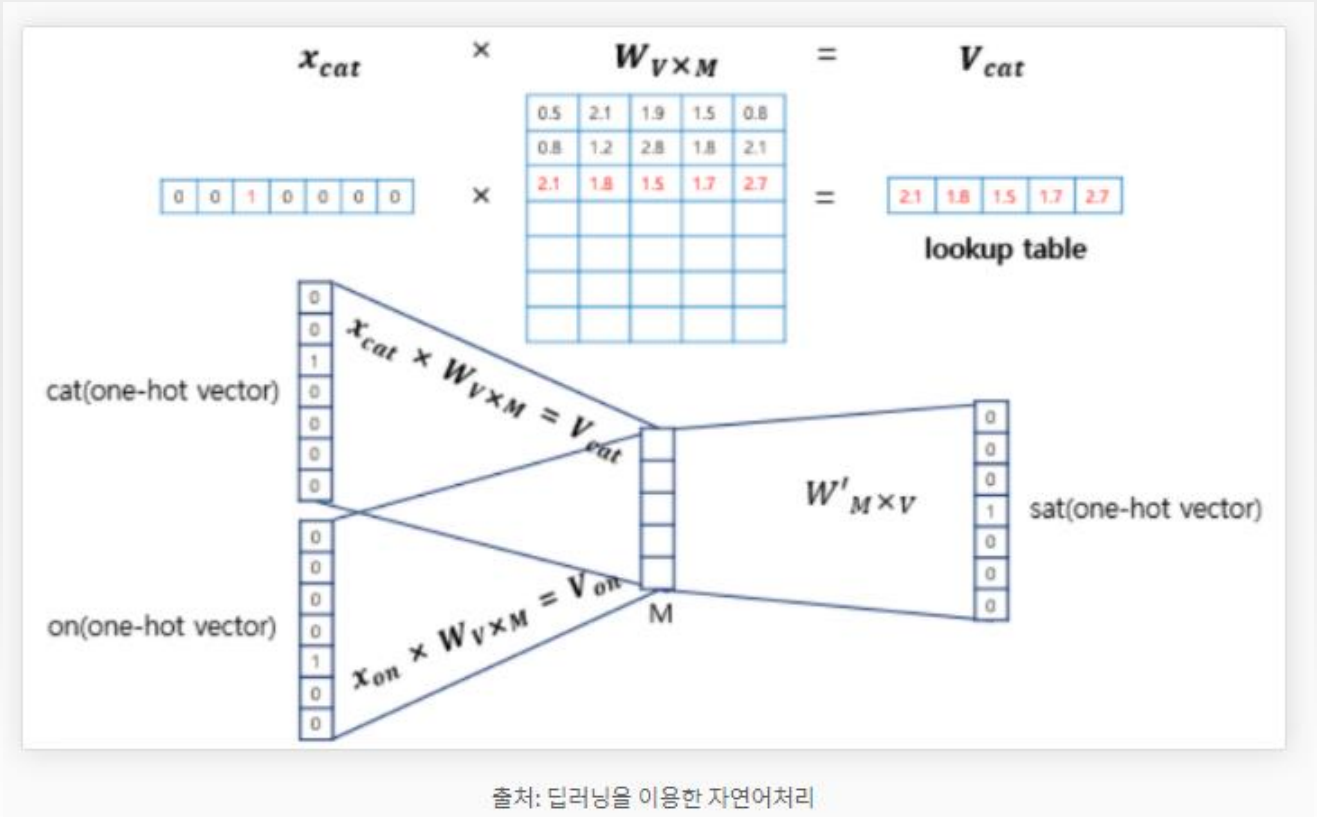


출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

- 맥락 단어의 원-핫 벡터를 x라 표기
  - 예) x_cat (cat에 대한 윗-핫 벡터)은 3번째 인덱스만 1이고 나머지는 다 0임
- 이와 가중치 행렬 W를 곱해주면 W의 3행에 해당하는 벡터가 도출
  - 가중치 행렬 W에서 '입력 벡터에서 1이 포함된 인덱스'에 해당하는 행을 추출하는 작업일 뿐
  - 즉, 원-핫 벡터 x에서 1의 값을 가지고 있는 인덱스를 i라 할 때, 가중치 행렬 W의 i번째 행을 가져오는 작업임

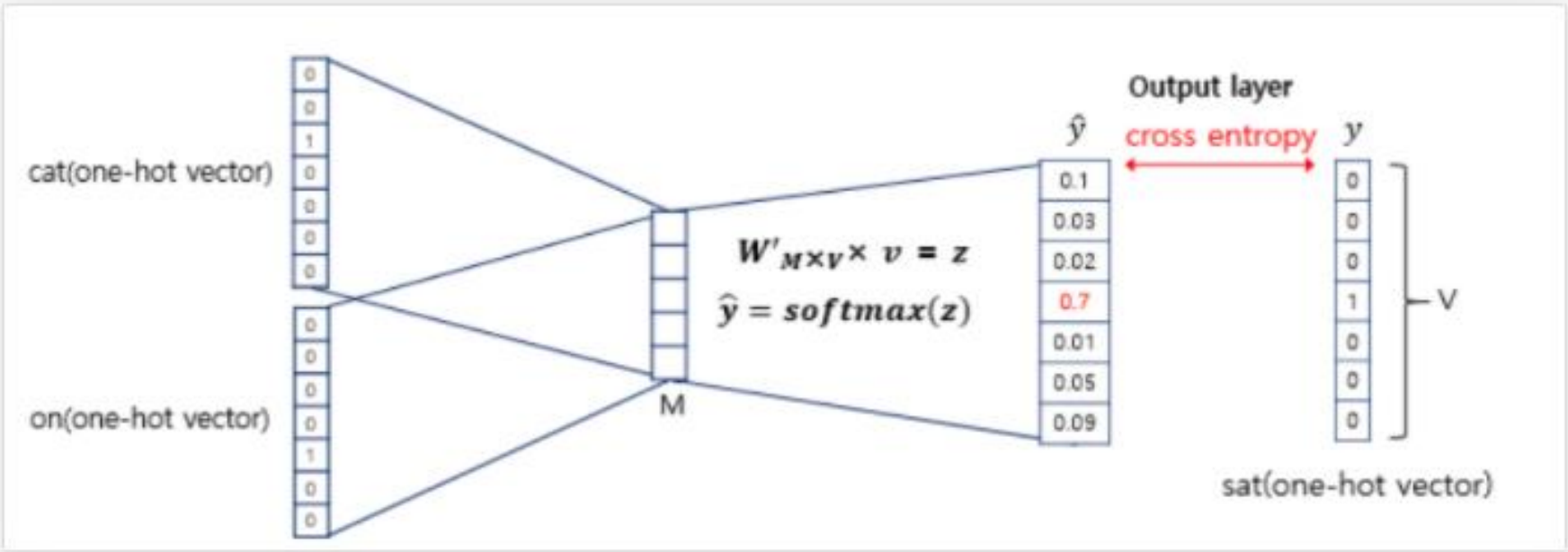


출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Word2Vec 프로세스 (CBOW 기반)]

- 앞에서 구한 평균 벡터 v를 출력층 가중치 행렬 W'와 곱합(곱해서 얻어진 z 벡터의 크기는 V (즉 7))
  - 이 벡터 z에 소프트맥스(Softmax)를 취해주면 확률 값에 해당하는 벡터 값이 구해짐
  - 소프트맥스를 취해주면 모든 원소의 합이 1인 상태로 바뀌기 때문에 확률을 나타내는 것

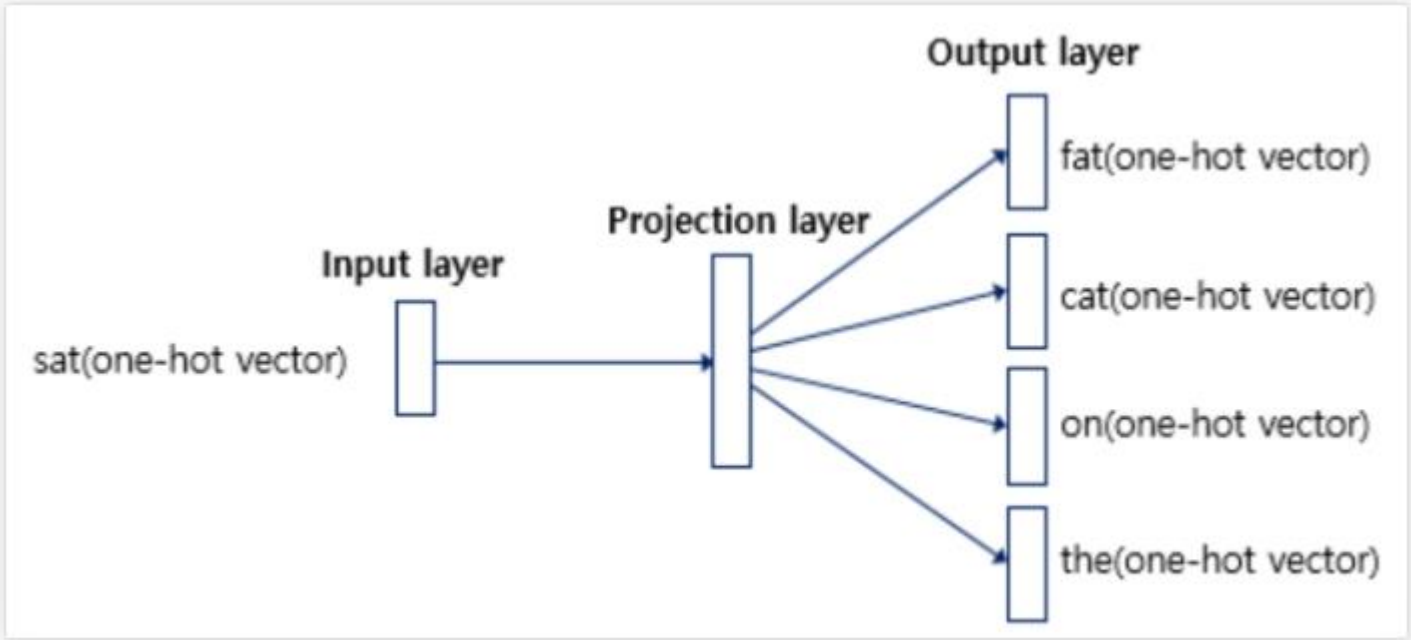


출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[Skip-gram]

- Skip-gram은 CBOW의 반대
- CBOW가 맥락 단어로부터 타깃 단어를 예측했다면 Skip-gram은 타깃 단어로부터 맥락 단어를 예측
- Skip-gram이 CBOW보다 성능이 우수하기 때문에 보통 Word2Vec을 쓴다고 하면 Skip-gram 임



출처: 딥러닝을 이용한 자연어처리

워드투벡터(Word2Vec)

[실습]

```
import nltk
nltk.download('movie_reviews')

from nltk.corpus import movie_reviews
sentences = [list(s) for s in movie_reviews.sents()]
sentences[0]

from gensim.models.word2vec import Word2Vec
%time

model = Word2Vec(sentences)

model.init_sims(replace=True)
model.wv.similarity('actor', 'actress')
model.wv.similarity('he', 'she')
model.wv.similarity('actor', 'she')
model.wv.most_similar('mother')
```