환경 설정

In [ ]:
```
from google.colab import drive
```

In [ ]:
```
drive.mount('/content/drive')
```

In [ ]:
```
# matplotlib 한글 폰트
!sudo apt-get install -y fonts-nanum
!sudo fc-cache -fv
!rm ~/.cache/matplotlib -rf
# 런타임 재시작
```

Mecab 설치

In [ ]:
```
! git clone https://github.com/SOMJANG/Mecab-ko-for-Google-Colab.git
```

In [ ]:
```
cd Mecab-ko-for-Google-Colab
```

In [ ]:
```
! bash install_mecab-ko_on_colab190912.sh
```

In [ ]:
```
import os

# # mecab-ko 설치
# os.chdir('/tmp/')
# !curl -LO https://bitbucket.org/eunjeon/mecab-ko/downloads/mecab-0.996-ko-0.9.1.tar.gz
# !tar zxfv mecab-0.996-ko-0.9.1.tar.gz
# os.chdir('/tmp/mecab-0.996-ko-0.9.1')
# !./configure
# !make
# !make check
# !make install

# mecab-ko-dic 설치
os.chdir('/tmp')
!curl -LO https://bitbucket.org/eunjeon/mecab-ko-dic/downloads/mecab-ko-dic-2.1.1-20180720.
tar.gz
!tar -zxvf mecab-ko-dic-2.1.1-20180720.tar.gz
os.chdir('/tmp/mecab-ko-dic-2.1.1-20180720')
!./autogen.sh
!./configure
!make
# !sh -c 'echo "dicdir=/usr/local/lib/mecab/dic/mecab-ko-dic" > /usr/local/etc/mecabrc'
!make install

# # mecab-python 설치: python3 기준
# os.chdir('/content')
# !git clone https://bitbucket.org/eunjeon/mecab-python-0.996.git
# os.chdir('/content/mecab-python-0.996')
# !python3 setup.py build
# !python3 setup.py install
```

In [ ]:

In [ ]:
```
import pandas as pd
import numpy as np
```

자연어 처리 라이브러리

1. NLTK
   - 대표적인 자연어처리 라이브러리
2. KoNLPy
   - 한글에 특화된 자연어처리 라이브러리
   - 토큰화, 형태소 분석
3. Gensim
   - 유사도 계산 등을 돕는 라이브러리
   - Topic Modeling: LDA, LSI, HDP
   - Word Embedding: word2vec
4. Scikit-learn
   - 대표적인 머신러닝 라이브러리이자 문서 전처리용 클래스 제공
   - Vectorizer

참고: 사회 혁신을 위한 데이터 중심의 서비스 기획자 ((http://hero4earth.com/blog/learning/2018/01/17/NLP_Basics_01/))

벡터가 어떻게 의미를 가지게 되는가

자연어 계산과 이해

- 컴퓨터로 자연어를 다루기 위해서는 임베딩을 활용한다.
  - 임베딩: 자연어를 숫자들의 벡터로 표현한 결과
- 자연어 의미를 임베딩에 함축하기 위해서 다음과 같은 세 가지 통계적 패턴 정보를 사용할 수 있다.

구분	BOW	언어모델	분포가정
내용	문장에 어떤 단어가 많이 쓰였는가	단어가 어떤 순서로 등장하는가	문장에서 어떤 단어들이 같이 나타났는가
대표통계량	TF-IDF	-	PMI
대표모델	Deep Averaging Network	ELMo, GPT	Word2Vec

- 이 세 가지 정보는 말뭉치의 통계적 패턴을 서로 다른 각도에서 분석하는 것이며 상호 보완적으로 사용된다.

어떤 단어가 많이 쓰였는가 (Bag of Words)

- 저자의 의도는 단어 사용 여부나 빈도에서 드러난다.
- 주제가 비슷한 문서라면 단어 빈도가 비슷할 것이다.

Bag of Words (BOW)

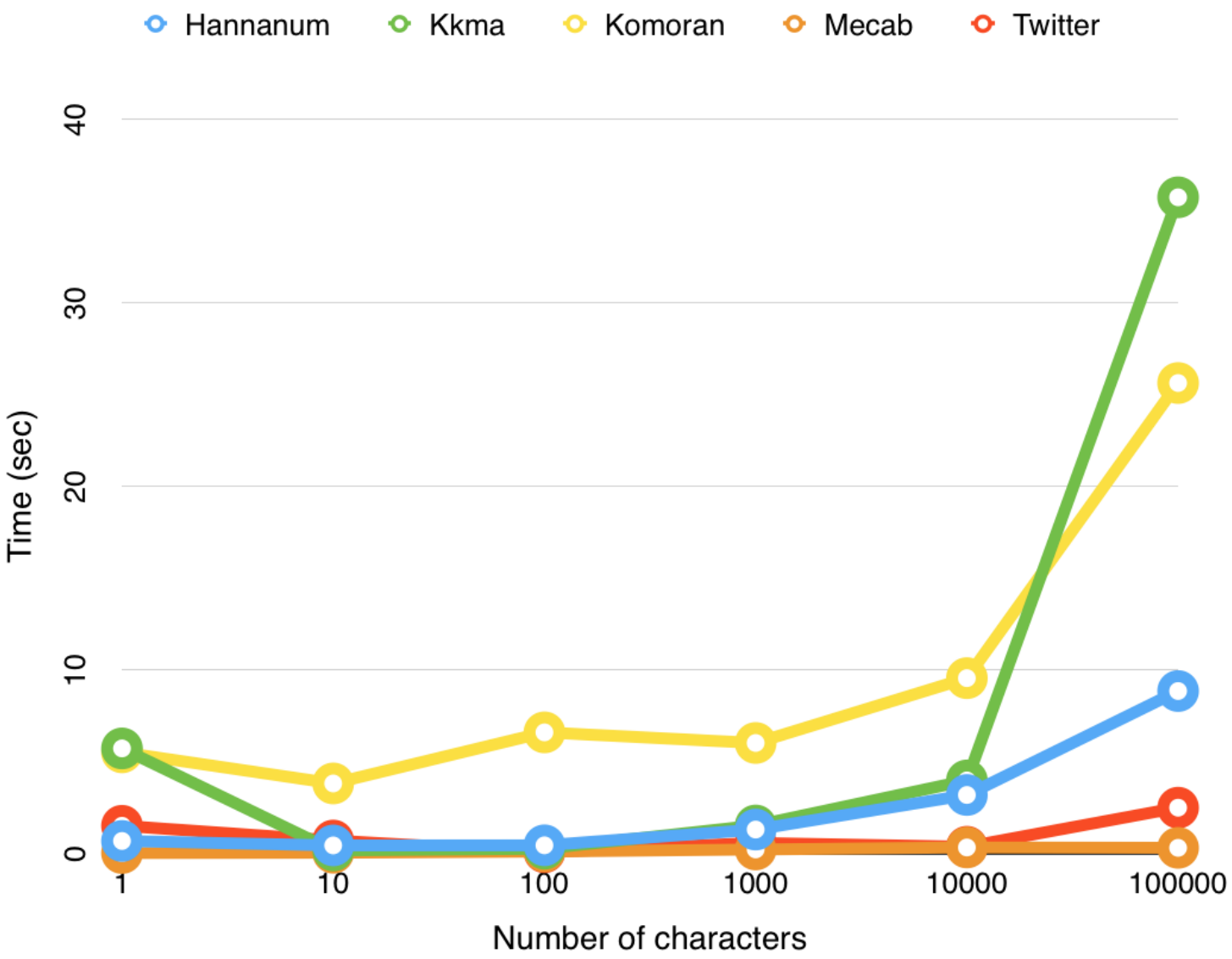
- 단어의 등장 순서에 관계없이 문서 내 단어의 등장 빈도를 임베딩으로 쓰는 기법
  - Bag: 순서를 고려하지 않고 중복 원소를 허용한 집합
- 경우에 따라 등장 여부를 임베딩으로 사용하기도 함
- 예시:

구분	메밀꽃 필 무렵	운수 좋은 날	사랑 손님과 어머니	삼포 가는 길
기차	0	2	10	7
막걸리	0	1	0	0
선술집	0	1	0	0

- 활용: **정보 검색** 분야에서 사용자의 질의에 가장 적절한 문서를 보여줄 때 질의를 BOW 임베딩으로 변환하고 질의와 검색 대상 문서 임베딩 간 코사인 유사도를 구해 유사도가 가장 높은 문서를 사용자에게 노출

In [ ]:
```
from konlpy.tag import Mecab
import re
mecab=Mecab()

# 정규 표현식을 통해 특수문자를 제거한다.
token=re.sub("[-=+,#/₩?:^$.@*₩"※~&%・!』₩₩ '|₩(₩)₩[₩]₩<₩>`₩'…》]","","정부가 발표하는 물가상승률과 소비자가 느끼는 물가상승률은 다르다.")

# mecab 형태소 분석기를 이용해 토큰화 작업을 진행한다.
token=mecab.morphs(token)
print(token)
```

- **Mecab?**
  - Mecab을 사용한 이유는 다른 한글 형태소 분석기에 비해 빠른 속도로 진행되어 문자의 개수가 많을 때도 비교적 빠르게 분석을 진행하는 편이기 때문.



- 성능에 관한 비교분석은 출처 (https://mr-doosun.tistory.com/22)를 참고함.

- **토큰화?**
  - 토큰화는 이후에 설명하겠지만 형태소에 따라 문장을 나눠줌.
  - 위 문장에서는 모든 특수문자를 제거했으나 사실 단순 제외해서는 안됨.
  - 예를 들어 'Q&A'라던가 '$100' 등 특수문자를 제거했을 때 그 의미가 변할 수 있기 때문

In [ ]:
```
word2index={}  # 단어들의 인덱스 번호를 저장한다.
bow=[]  # 단어들의 출현 횟수를 센다.

for voca in token:
    if voca not in word2index.keys():
        # token에 있는 단어들 중 word2index에 없는 단어를 새로 추가한다.
        word2index[voca]=len(word2index) # index 번호
        # 단어는 최소 1번 이상 등장했기 때문에 bow에 1을 넣어준다.
        bow.insert(len(word2index)-1,1)
    else:
        # word2index에 있는 단어(재등장)의 경우 단어의 개수에 +1을 해준다.
        index=word2index.get(voca)
        bow[index]=bow[index]+1

print(word2index)
```

In [ ]:
```
for i in range(len(word2index.keys())):
    print('%s : %d 번' % (list(word2index.keys())[i], bow[i]))
```

- 명사만 추출하여 분석을 진행하는 경우도 많다.

In [ ]:
```
token=re.sub("[-=+,#/₩?:^$.@*₩"※~&%・!』₩₩ '|₩(₩)₩[₩]₩<₩>`₩'…》]","","정부가 발표하는 물가상승률과 소비자가 느끼는 물가상승률은 다르다.")

token=mecab.nouns(token)

word2index={}
bow=[]

for voca in token:
    if voca not in word2index.keys():
        word2index[voca]=len(word2index)
        bow.insert(len(word2index)-1,1)
    else:
        index=word2index.get(voca)
        bow[index]=bow[index]+1

for i in range(len(word2index.keys())):
    print('%s : %d 번' % (list(word2index.keys())[i], bow[i]))
```

TF-IDF(Term Frequency-Inverse Document Frequency)

- 등장 배경:
  - 조사 같은 경우는 등장 빈도가 높지만 문서의 주제를 제대로 표현하는 단어라고 하기 어려움
  - 이를 극복하기 위해 등장 빈도가 높은 단어들의 가중치를 낮추어 표현하는 기법인 TF-IDF 등장
- 단어-문서 행렬(Document Term Matrix; DTM)에 가중치를 계산하여 값을 부여
  - 어떤 단어의 주제 예측력(해당 단어를 통해 문서의 주제를 가늠할 수 있는 정도)가 강할 수록 가중치가 커지도록 아래와 같이 수식 설계

$$TfIdf(w) = TF(w) \times log(\frac{N}{DF(w)})$$

  - TF: 어떤 단어가 특정 문서에 얼마나 많이 쓰였는지 단순 빈도를 나타낸다.
  - DF: 특정 단어가 나타난 문서의 수
  - IDF: 전체 문서 수(N)을 해당 단어의 DF로 나눈 뒤 로그를 취한 값
  - 단어의 TF가 높으면 TF-IDF 값이 커지고, 단어가 많은 문서에 등장할 수록(DF가 커지면) TF-IDF 값이 작아짐

- DTM: 다수의 문서에서 등장하는 단어들의 빈도를 행렬로 표현한 것
  - BOW를 하나의 행렬로 만든 것이라 생각할 수 있음

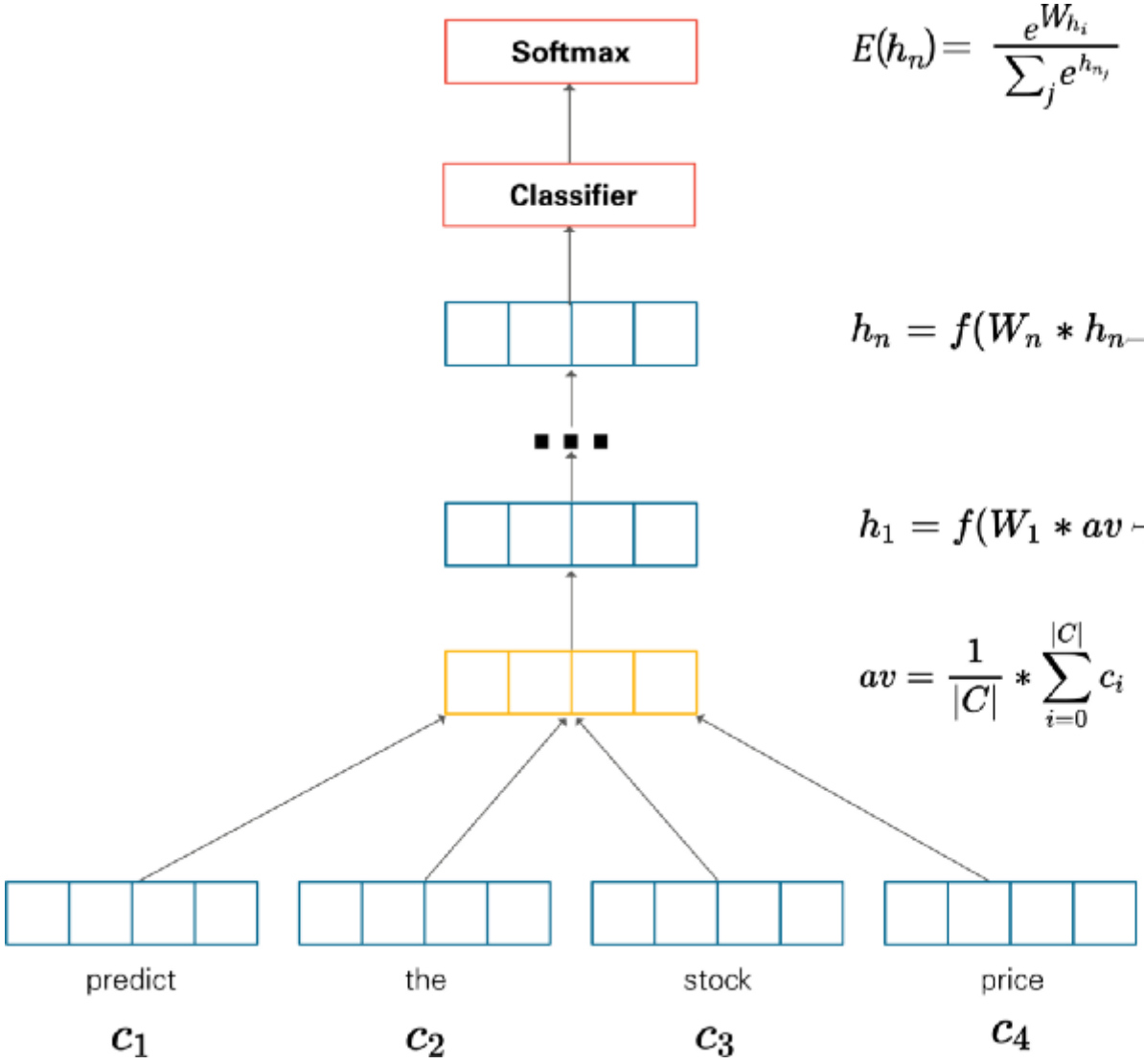
In [ ]:
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
```

In [ ]:
```
docs = ['이른 아침 작은 새들 노랫소리 들려오면 언제나 그랬듯 아쉽게 잠을 깬다',
        '창문 하나 햇살 가득 눈부시게 비쳐오고 서늘한 냉기에 재채기할까 말까',
        '가을 아침 내겐 정말 커다란 기쁨이야 가을 아침 내겐 정말 커다란 행복이야']

# Term Frequency만 사용한 경우
tf = CountVectorizer()
dtm = tf.fit_transform(docs)

dt = pd.DataFrame(dtm.toarray())
dt.columns = tf.get_feature_names()
dt
```

In [ ]:
```
# Tf-idf를 사용한 경우
tfidf = TfidfVectorizer()
dtm = tfidf.fit_transform(docs)

dt = pd.DataFrame(dtm.toarray())
dt.columns = tfidf.get_feature_names()
dt
```

Deep Averaging Network(DAN)

- BOW 가정의 뉴럴 네트워크 버전



$$E(h_n) = \frac{e^{W_{h_i}}}{\sum_j e^{h_{n_j}}} \quad (4)$$

$$h_n = f(W_n * h_{n-1} + b_n) \quad (3)$$

$$h_1 = f(W_1 * av + b_1) \quad (2)$$

$$av = \frac{1}{|C|} * \sum_{i=0}^{|C|} c_i \quad (1)$$

1. 단어의 순서를 고려하지 않고, 단어의 임베딩 전체의 평균을 계산
   - dropout-inspired regularization technique을 사용하면 성능이 더 좋아짐
2. 평균값은 히든 레이어로 보냄
3. 결과값을 통해 분류

- 간단하지만 성능이 좋아 현업에서 분류 문제에 많이 사용

참고:

- A Method for Building a Strong Baseline Text Classifier (https://medium.com/@dhoeschele/a-method-for-building-a-strong-baseline-text-classifier-53451ee6c250)
- Pytorch Deep Average Network as Baseline (https://www.kaggle.com/bobazooba/pytorch-deep-average-network-as-baseline)

단어가 어떤 순서로 쓰였는가 (언어 모델)

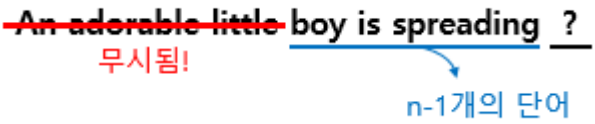
- 단어의 **등장 순서**를 학습해 주어진 단어 시퀀스가 얼마나 자연스러운지 확률 부여

통계 기반 언어 모델

- 단어가 n개 주어질 경우 n개 단어가 동시에 나타날 확률 $P(w_1, w_2, \ldots, w_n)$을 반환
- 조건부 확률을 활용해 최대우도추정법(MLE)으로 표현하면 다음과 같음

$$P(w_5|w_1, w_2, w_3, w_4) = \frac{Freq(w_1, w_2, w_3, w_4, w_5)}{Freq(w_1, w_2, w_3, w_4)}$$

- 위의 식에서 단어들이 한 번도 등장하지 않는다면 분자가 0이 될 수 있음 → 희소 문제(Sparsity Problem) 발생
- 희소 문제를 완화하기 위해 n-gram(n개의 단어를 묶은 것)을 사용
  - 직전 $n-1$개의 단어를 사용하여 전체 단어 시퀀스 등장 확률을 근사하는 방법



  - 한 상태의 확률은 그 직전 상태에만 의존한다는 마코프(Markov) 가정에 기반한 것
  - 바이그램을 사용한다면 전체 단어의 등장 확률을 아래와 같이 수정하여 표현할 수 있음

$$P(w_1, w_2, w_3, w_4, w_5)$$
$$\approx P(w_1) \times P(w_2|w_1) \times P(w_3|w_2) \times P(w_4|w_3) \times P(w_5|w_4)$$

  - n-gram을 사용하여 모델의 식을 수정하면 아래와 같음

$$P(w_n|w_{n-1}) = \frac{Freq(w_n, w_{n-1})}{Freq(w_{n-1})}$$

  - tradeoff: n을 크게 선택하면 희소 문제가 커지고 모델 사이즈가 커짐. 작게 선택하면 근사의 정확도는 현실과 멀어짐. 따라서 **최대 5를 넘지 말기를 권장**. (출처: 딥러닝을 이용한 자연어처리 입문 (https://wikidocs.net/21692))

In [ ]:
```
# https://lovit.github.io/nlp/2018/10/23/ngram/ 를 참고하여 수정함

from collections import defaultdict

def get_ngram_counter(docs, min_count=10, n_range=(1,3)):

    def to_ngrams(words, n):
        ngrams = []
        for b in range(0, len(words) - n + 1):
            ngrams.append(tuple(words[b:b+n]))
        return ngrams

    n_begin, n_end = n_range
    ngram_counter = defaultdict(int)
    for doc in docs:
        words = mecab.pos(doc, join=True)
        for n in range(n_begin, n_end + 1):
            for ngram in to_ngrams(words, n):
                ngram_counter[ngram] += 1

    ngram_counter = {
        ngram:count for ngram, count in ngram_counter.items()
        if count >= min_count
    }

    return ngram_counter

class NgramTokenizer:

    def __init__(self, ngrams, base_tokenizer, n_range=(1, 3)):
        self.ngrams = ngrams
        self.base_tokenizer = base_tokenizer
        self.n_range = n_range

    def __call__(self, sent):
        return self.tokenize(sent)

    def tokenize(self, sent):
        if not sent:
            return []

        unigrams = self.base_tokenizer.pos(sent, join=True)

        n_begin, n_end = self.n_range
        ngrams = []
        for n in range(n_begin, n_end + 1):
            for ngram in self.to_ngrams(unigrams, n):
                ngrams.append(' - '.join(ngram))
        return ngrams

    def to_ngrams(self, words, n):
        ngrams = []
        for b in range(0, len(words) - n + 1):
            ngrams.append(tuple(words[b:b+n]))
        return ngrams
```

In [ ]:
```
ngram_counter = get_ngram_counter(docs, min_count=2) # 2번 이상 등장
ngram_tokenizer = NgramTokenizer(ngram_counter, mecab)
```

In [ ]:
```
ngram_tokenizer(docs[0])
```

In [ ]:
```
vectorizer = CountVectorizer(
    tokenizer = ngram_tokenizer,
    lowercase = False,
)
x = vectorizer.fit_transform(docs)
x.shape
```

In [ ]:
```
ngram_counter.items()
```

In [ ]:
```
for ngram, count in sorted(ngram_counter.items(), key=lambda x:-x[1]):
    if str(ngram).find('가을/NNG') != -1: # 가을을 포함한 경우
        print(ngram, count)
```

- n-gram 모델을 사용하더라도 단어가 한 번도 등장하지 않을 경우 조건부 확률의 값이 0이 되어버리기 때문에 백오프, 스무딩 등의 기법이 제안됨
  - 백오프: n-gram 등장 빈도를 n보다 작은 범위의 단어 시퀀스 빈도로 근사
  - 스무딩: n-gram 등장 빈도에 k를 더해줌 (add-K 스무딩)
    - K가 1인 경우 라플라스(Laplace) 스무딩

뉴럴 네트워크 기반 언어 모델 (6장)

- 주어진 단어 시퀀스로 다음 단어를 예측하는 과정에서 학습됨
  - 학습 완료 후 결과물을 단어나 문장의 임베딩으로 활용 ex ELMo, GPT 등
- 마스크 언어 모델(Masked language model): 문장 중간(마스크)에 어떤 단어가 올지 예측하는 과정에서 학습됨 ex BERT
  - 문장 전체를 다 보고 중간에 있는 단어를 예측하는, 양방향 학습이 가능
  - 순차적으로 단어를 입력받아 다음 단어를 맞추는 기존의 일방향 언어 모델보다 품질이 좋음

어떤 단어가 같이 쓰였는가 (분포 가정)

- 단어의 의미는 **주변 문맥(context)**를 통해 유추할 수 있다.

분포 가정

- 자연어 처리에서의 분포: 윈도우 내에 동시에 등장하는 이웃 단어 또는 문맥의 집합

분포와 의미

형태소 관점

- 형태소: 의미를 가지는 최소 단위
- 형태소를 분석하는 방법: 계열 관계
  - 해당 형태소 자리에 다른 형태소가 대치되어 사용될 수 있는가
  - "*철수*가 *밥*을 먹었다."라는 문장에서 철수와 밥 자리에 다른 단어가 대치될 수 있다면 이 둘은 형태소임

품사 관점

- 품사: 단어를 문법적 성질의 공통성에 따라 언어학자들이 몇 갈래로 묶어 놓은 것
- 품사 분류 기준: 기능(한 단어가 문장 가운데서 다른 단어와 맺는 관계), 의미(단어의 형식적 의미), 형식(단어의 형태적 특징)
  - 이 중 가장 중요한 기준은 기능

점별 상호 정보량(PMI)

- 두 확률변수 사이의 상관성을 계량화하는 단위
- 두 확률변수가 독립인 경우 PMI 값은 0

$$PMI(A,B) = log\frac{P(A,B)}{P(A) \times P(B)}$$

- 분포 가정에 따라 단어 가중치를 할당하고, 행렬의 행 벡터 자체를 단어의 임베딩으로 사용할 수 있음

In [ ]:
```
from nltk import collocations
```

In [ ]:
```
# 참고: https://konlpy-ko.readthedocs.io/ko/v0.4.3/examples/collocations/

doc = ' '.join(docs)
print(doc)

measures = collocations.BigramAssocMeasures()

print('₩nCollocations among tagged words:')
tagged_words = mecab.pos(doc)
finder = collocations.BigramCollocationFinder.from_words(tagged_words)
print(finder.nbest(measures.pmi, 5)) # top 5 n-grams with highest PMI

print('₩nCollocations among words:')
words = [w for w, t in tagged_words]
ignored_words = [u'']
finder = collocations.BigramCollocationFinder.from_words(words)
finder.apply_word_filter(lambda w: len(w) < 2 or w in ignored_words)
finder.apply_freq_filter(2) # only bigrams that appear 2+ times
print(finder.nbest(measures.pmi, 5))
```

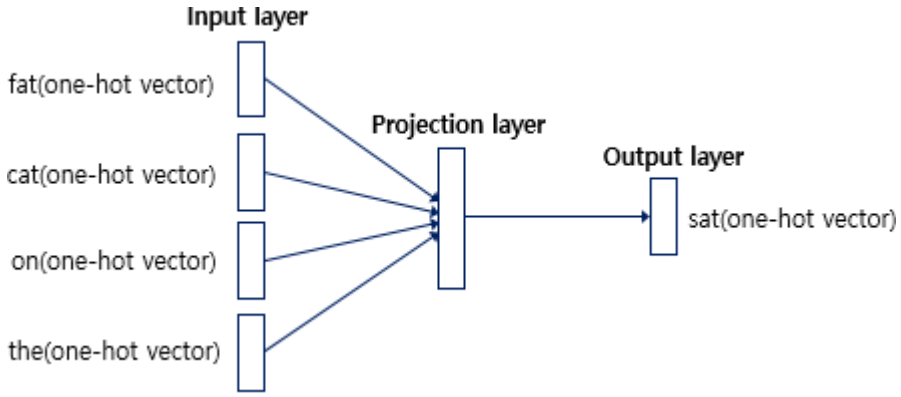
Word2Vec (4장)

- 분포 가정의 대표적인 모델이며 PMI 행렬과 깊은 연관이 있음 (ex 잠재의미분석)
- 분산 표현을 통해 저차원에 단어의 의미를 여러 차원에 분산하여 표현함으로써 단어간 유사도를 계산할 수 있음

CBOW 모델

- 문맥 단어들을 통해 타깃 단어 하나를 맞추는 과정에서 학습

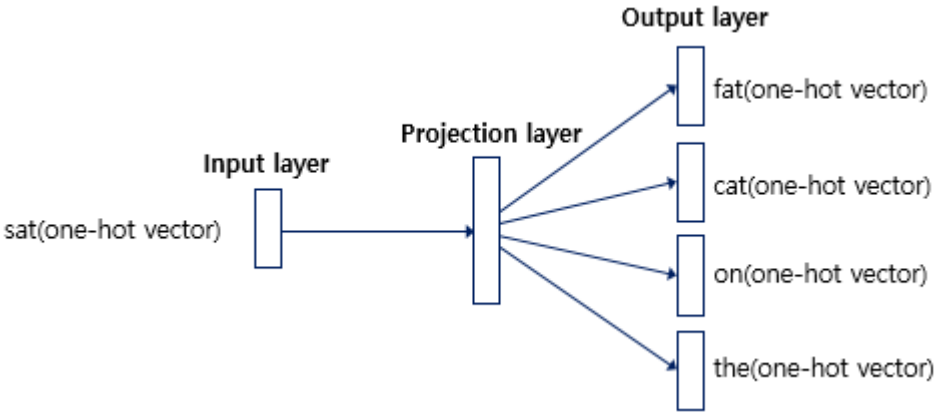
중심 단어	주변 단어
The fat cat sat on the mat	
[1, 0, 0, 0, 0, 0]	[0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0]
The fat cat sat on the mat	
[0, 1, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0]
The fat cat sat on the mat	
[0, 0, 1, 0, 0, 0]	[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 0,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



- 입력: 사용자가 정한 윈도우 범위 안에 있는 주변 단어들의 원-핫(one-hot) 벡터
- 출력: 예측하고자 하는 중간 단어의 원-핫 벡터
- 주변 단어의 평균 값을 갖고 예측함

Skip-gram 모델

- 타깃 단어를 통해 문맥 단어가 무엇일지 예측하는 과정에서 학습



- 중심 단어로부터 주변 단어를 예측
- 전반적으로 Skip-gram이 CBOW보다 성능이 좋은 편