

# CS1544

## 자연어처리

### 3. Word Embedding

*Since 1947*

**SEOKYEONG UNIVERSITY**

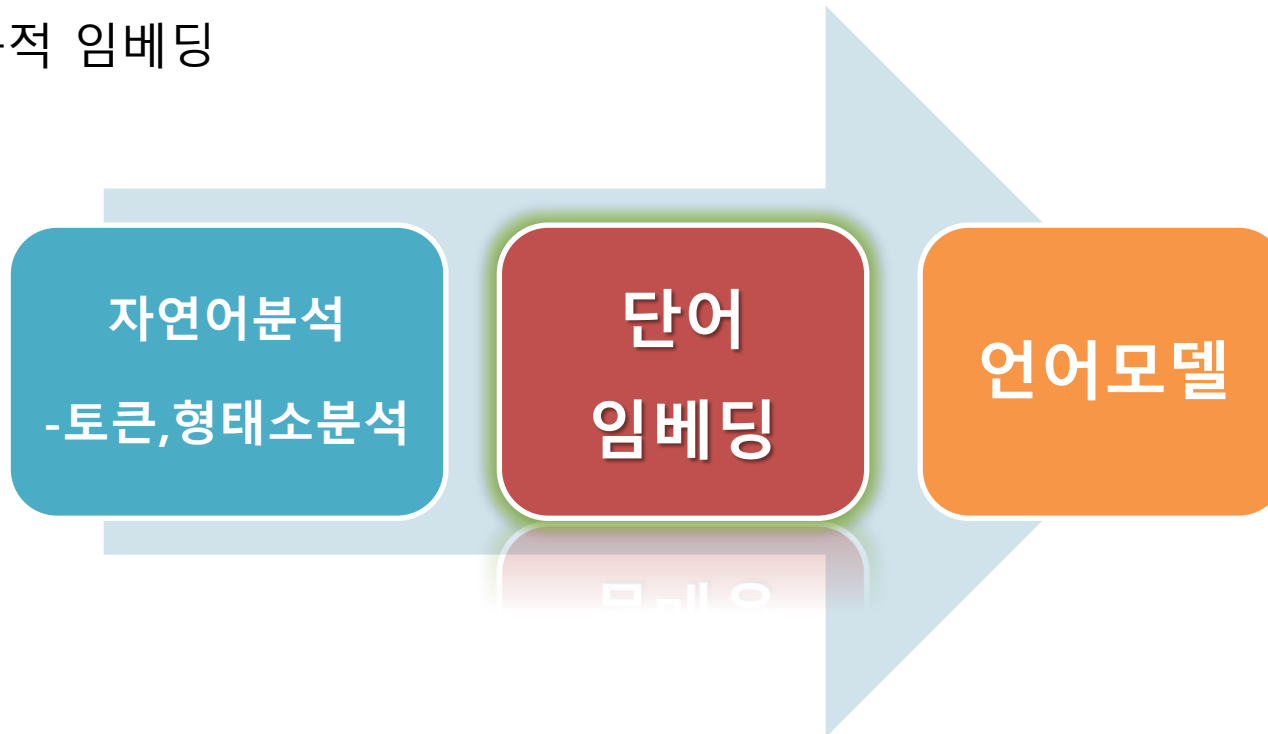
In accordance with the spirit of "Devotion to the Welfare of Mankind (Hong-Ik-In-Kan)" Seokyeong University is committed to delving deeply into advanced learning, teaching and training.

We hope that young people will grow to become competent leaders, and thus contribute to realization of an ideal society where the coprosperity of mankind is fully achieved.



## ❖ Word Embedding

- 초기 단어 표현
- 고정 임베딩
- 동적 임베딩





## ❖ 자연어 처리에서의 단어 표현

- 자연어 분석(분해) 단계에서 정리된 단어를 컴퓨터가 이해할 수 있는 수치형 값으로 표현하는 것
- 단순 "단어 표현"에서 의미가 반영된 "단어 임베딩"으로 발전
  - 단어 표현 모델: One-Hot Encoding, Bag-of-Words (BoW), TF-IDF
  - 단어 임베딩 모델: Word2Vec, GloVe, FastText, ELMo, BERT, GPT 계열



# 1. 단어 표현: (1) One-Hot Encoding

## ❖ One-Hot Encoding

- 단어를 고정된 크기의 벡터로 표현하며, 벡터 공간의 특정 위치에만 1을 주고, 나머지는 모두 0으로 표현하는 방식
  - 어휘집에 10,000 단어가 있으면, 한 단어는 10,000차원의 벡터로 표현됨.
  - 예) 'cat' = [0, 0, 1, 0, 0, ..., 0]  
'dog' = [0, 0, 0, 0, 1, ..., 0]
- 장점:
  - 단순하고 직관적이며 구현이 쉬움
  - 단어들을 고유하게 구분 가능
- 한계:
  - 차원이 너무 커짐(차원의 저주).
  - 희소 행렬 문제: 매우 희소(sparse)한 벡터 사용.
  - 의미적 유사성 반영 불가능 → "king"과 "queen"은 전혀 다른 벡터로 표현됨.
- *Python 라이브러리*
  - from sklearn.preprocessing import **OneHotEncoder**

# 1. 단어 표현: (2) Count based Encoding

## ❖ Bag-of-Words (BoW)

- 문서를 단어의 '가방(bag)'처럼 보고, 등장한 단어의 개수만 기록
- 단어 사전(단어 집합, vocabulary)을 만들고, 각 단어의 **출현 횟수**(Term Frequency, TF)를 세어 벡터로 변환
  - 예) 문서 집합 {"I love NLP", "You love Python"}
    - ☞ Vocabulary = {I, love, NLP, You, Python}
    - ☞ "I love NLP and I love Python !" = [2,2,1,0,1]
- 장점:
  - 간단하고 직관적, 빠른 계산
- 한계:
  - 출단어 순서 정보 손실, 희소 행렬 문제
- *Python 라이브러리*
  - from sklearn.feature\_extraction.text import **CountVectorizer**

# 1. 단어 표현: (2) Count based Encoding

## ❖ TF-IDF (Term Frequency-Inverse Document Frequency)

- BOW(TF)에서 단순 빈도의 한계를 보완, "중요 단어" 가중치를 부여하는 방법
- TF (Term Frequency): 특정 단어가 문서에서 얼마나 자주 등장하는지
- IDF (Inverse Document Frequency): 특정 단어가 여러 문서에서 자주 나오면 그 중요도를 낮춤

$$tf-idf(t, d) = tf(t, d) \times idf(t, d)$$

$$idf(t, d) = \log \frac{n_d}{1 + df(d, t)}$$

- 장점:
  - 불용어("the", "is") 중요도 낮춤
  - 가중치에 의해 의미 있는 단어 강조
- 한계:
  - 여전히 단어 순서 반영 불가
  - 단어 의미 자체는 고려하지 않음

### ■ Python 라이브러리

- `from sklearn.feature_extraction.text import TfidfVectorizer`

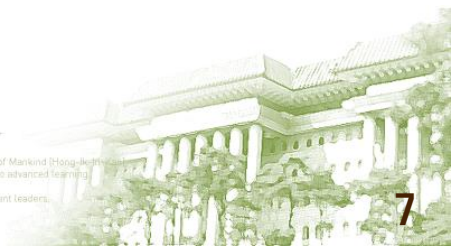


## ❖ Anaconda Powershell 실행

- 1) 작업폴더(NLP\_PJT)로 디렉토리 변경: `cd C:\Users\OOOOW\NLP_PJT`
- 2) 가상 환경 활성화: `.\NLP_Lec_env\Scripts\activate`
- 3) jupyter notebook 실행: `Start-Process jupyter notebook`

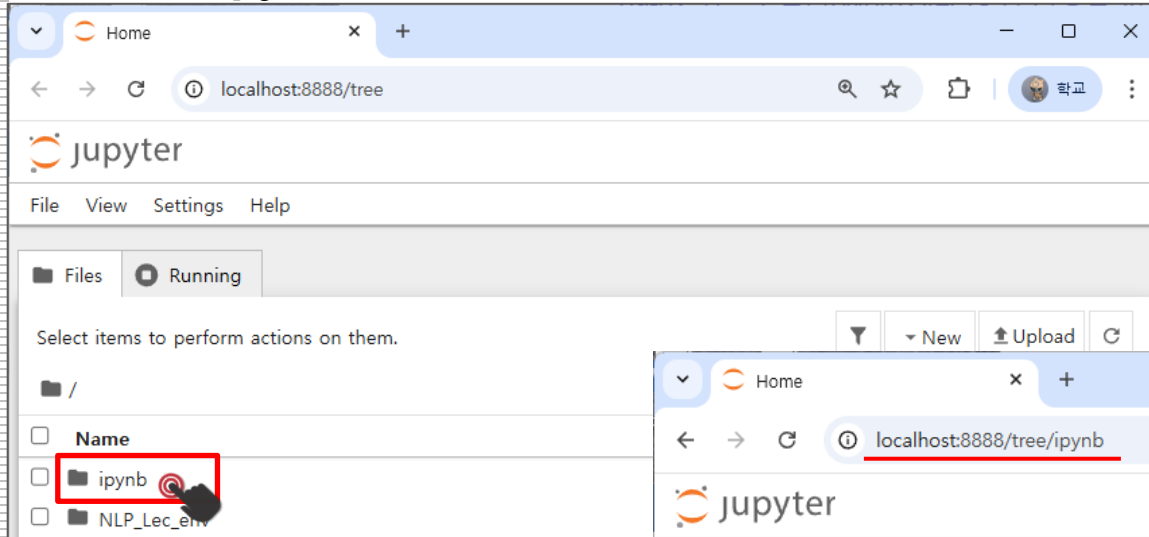
```
Anaconda Powershell Prompt

(base) PS C:\Users\User> cd NLP_PJT
(base) PS C:\Users\User\NLP_PJT> .\NLP_Lec_env\Scripts\activate
(NLP_Lec_env) (base) PS C:\Users\User\NLP_PJT> Start-Process jupyter notebook
(NLP_Lec_env) (base) PS C:\Users\User\NLP_PJT> _
```

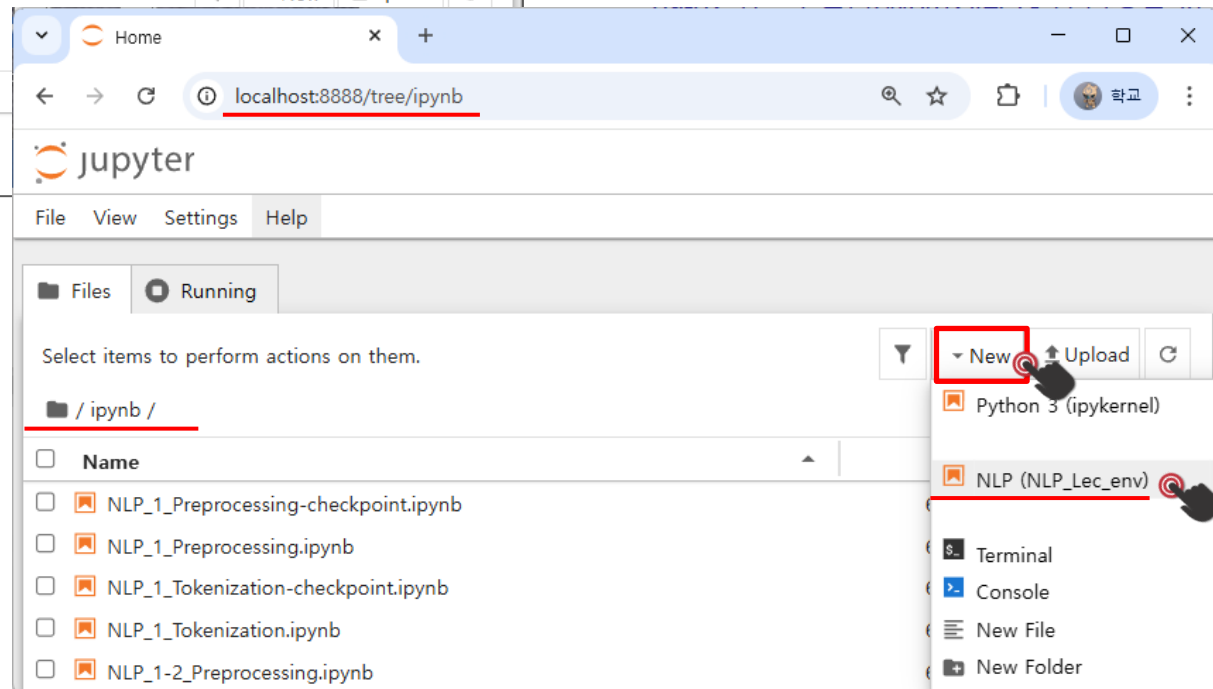


## ❖ jupyter notebook 새 파일 생성

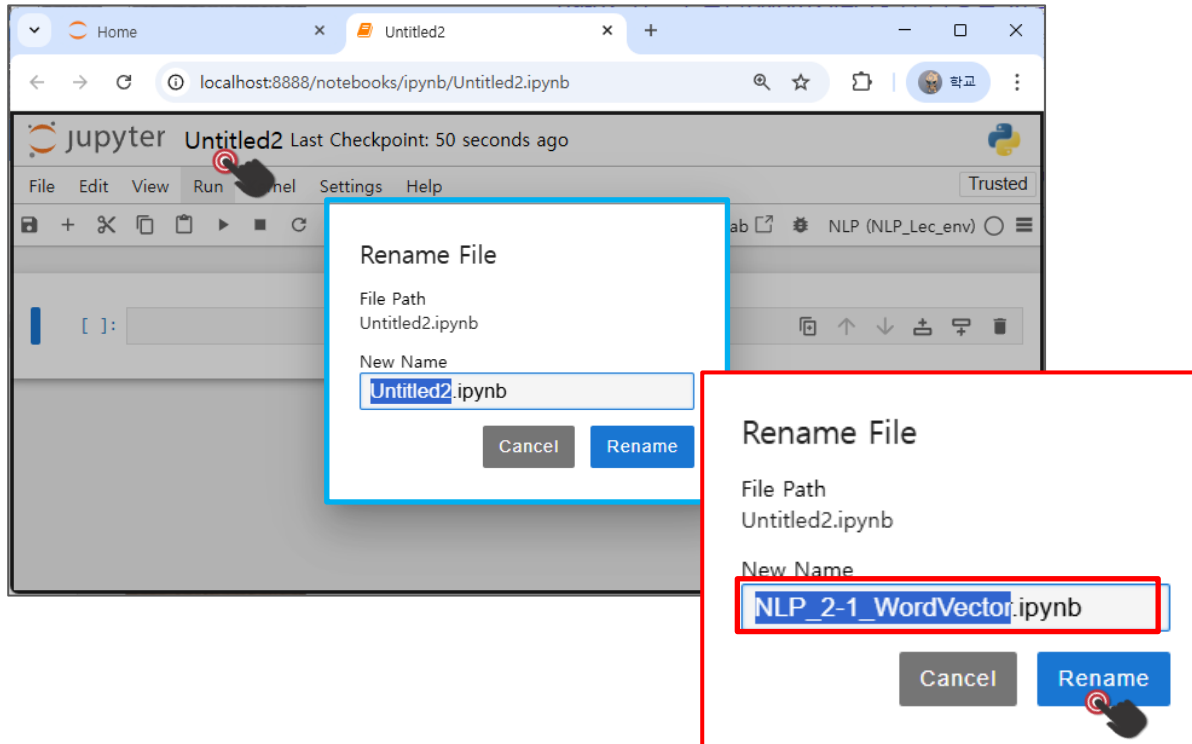
### ■ 1) [ipynb] 폴더 클릭



### ■ 2) [New] – NLP 선택 (가상환경 커널 선택)



## 3) 파일 이름 저장





```
[1]: corpus = [
    "I love NLP",
    "NLP is fun",
    "I love machine learning"
]
```

## 1. One-Hot Encoding

```
[2]: from sklearn.preprocessing import OneHotEncoder
import numpy as np
```

```
[3]: # 단어 목록 생성
words = set()

for sent in corpus:
    for word in sent.split():
        words.add(word)

word_list = sorted(list(words))
```

```
[4]: print(word_list)

['I', 'NLP', 'fun', 'is', 'learning', 'love', 'machine']
```





```
[5]: # 원-핫 인코더 생성
encoder = OneHotEncoder(sparse_output=False)
encoder.fit(np.array(word_list).reshape(-1,1))
```

OneHotEncoder의 (샘플 수, 특성 수)의 2차원 입력 형태로 맞추기 위해 reshape

```
[5]: OneHotEncoder
OneHotEncoder(sparse_output=False)
```

```
[6]: # 원-핫 인코딩
for sent in corpus:
    encoded = encoder.transform(np.array(sent.split()).reshape(-1,1))
    print(f"\n문장: {sent}")
    print(encoded)
```

```
문장: I love NLP
[[1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 0.]]
```

```
문장: NLP is fun
[[0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]]
```

```
문장: I love machine learning
[[1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 1. 0. 0.]]
```



```
[7]: encoded_sent = encoder.transform(np.array("I love fun NLP".split()).reshape(-1,1))  
print(f"\n새로운 문장: {sent}")  
print(encoded)
```

새로운 문장: I love machine learning

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



## ▼ ref. 원-핫 인코더 - 희소행렬 ¶

```
[8]: # 원-핫 인코더 - 기본값: 희소행렬
encoder2 = OneHotEncoder()
encoder2.fit(np.array(word_list).reshape(-1,1))

for sent in corpus:
    X_sparse = encoder2.transform(np.array(sent.split()).reshape(-1,1))

    print(f"\n문장: {sent}")
    print("원-핫 인코딩 행렬 형태:", type(X_sparse))
    print("희소행렬 크기:", X_sparse.shape)
    print(X_sparse)
```

문장: I love NLP

원-핫 인코딩 행렬 형태: <class 'scipy.sparse.\_csr.csr\_matrix'>

희소행렬 크기: (3, 7)

(0, 0)	1.0
(1, 5)	1.0
(2, 1)	1.0

문장: NLP is fun

원-핫 인코딩 행렬 형태: <class 'scipy.sparse.\_csr.csr\_matrix'>

희소행렬 크기: (3, 7)

(0, 1)	1.0
(1, 3)	1.0
(2, 2)	1.0

문장: I love machine learning

원-핫 인코딩 행렬 형태: <class 'scipy.sparse.\_csr.csr\_matrix'>

희소행렬 크기: (4, 7)

(0, 0)	1.0
(1, 5)	1.0
(2, 6)	1.0
(3, 4)	1.0





## 2. BoW (CountVector)

```
[9]: corpus = [
    "I love NLP",
    "NLP is fun",
    "I love NLP, You love NLP! "
]
```

```
[10]: from sklearn.feature_extraction.text import CountVectorizer
```

```
[11]: #CountVector 객체 생성
vectorizer = CountVectorizer()
```

```
[12]: #CountVector 설정 및 수행
X_bow = vectorizer.fit_transform(corpus)
```

```
[13]: print("Bow Vocabulary:", vectorizer.vocabulary_)
print("Bow Representation:\n", X_bow.toarray())
```

Bow Vocabulary: {'love': 2, 'nlp': 3, 'is': 1, 'fun': 0, 'you': 4}

Bow Representation:

```
[[0 0 1 1 0]
 [1 1 0 1 0]
 [0 0 2 2 1]]
```



### 3. TF-IDF (tfidf Vector)

```
[14]: from sklearn.feature_extraction.text import TfidfVectorizer
```

```
[15]: tfidf = TfidfVectorizer()  
X_tfidf = tfidf.fit_transform(corpus)
```

```
[16]: print("TF-IDF Vocabulary:", tfidf.vocabulary_)  
print("TF-IDF Representation:\n", X_tfidf.toarray())
```

TF-IDF Vocabulary: {'love': 2, 'nlp': 3, 'is': 1, 'fun': 0, 'you': 4}

TF-IDF Representation:

```
[[0.          0.          0.78980693 0.61335554 0.          ]  
 [0.65249088 0.65249088 0.          0.38537163 0.          ]  
 [0.          0.          0.70094487 0.54434622 0.46082913]]
```

### ❖ 기존 단어 표현(단순 벡터화)의 한계

- 희소성 문제
  - 단순 벡터 방식(BoW, TF-IDF)은 문서의 차원이 단어 수(수만~수십만 차원)이므로 희소 행렬의 문제 발생
- 의미 표현 불가
- 일반화의 문제
  - 훈련 데이터에 없는 새로운 단어는 표현 불가능

### ❖ 단어 임베딩 (Word Embedding)

- 단어를 고정된 길이의 실수 벡터로 표현하는 방법
- 단어 간의 의미적 유사성을 벡터 공간 상의 거리·방향으로 표현
- 기존의 단순 빈도 기반 표현(BOW, TF-IDF)과 달리, 단어의 의미와 맥락(context)까지 반영한 벡터화
  - 단어를 단순히 'ID'로 처리하는 것이 아니라, '의미 있는 수학적 점'으로 처리



### ❖ 단어 임베딩 장점

- 희소성 문제 해결
  - 단어 임베딩은 50~300차원 수준의 조밀한(dense) 벡터로 표현하여 계산 효율성 향상
- 의미 유사성 표현
  - 의미상 유사한 단어들을 벡터 공간에서 가까운 위치에 매핑
- 일반화 능력
  - 훈련 데이터에 없는 새로운 문장에도 **비슷한 맥락 학습 효과** 적용 가능



## ❖ 분포 가설(Distributional Hypothesis)

- "같은 맥락에서 쓰이는 단어들은 비슷한 의미를 가진다." (Harris, 1954)
- ☞ 단어의 의미는 주변 단어와의 분포로 파악할 수 있음

## ❖ 분산 표현(Distributed Representation)

- 단어 의미를 여러 차원에 분산해서 표현하며, 각 차원은 단어의 의미적 속성을 잠재적으로 반영.
- 단어를 저차원의 밀집 벡터(Dense Vector)로 나타내고, 단어 간 의미적 유사성을 벡터 연산으로 표현, 벡터 연산을 통해 의미 관계 추론 가능.
  - 단어 벡터 간 연산으로 단어 의미 관계 표현 가능
    - 예시) king - man + woman  $\approx$  queen

## ❖ 단어 간 유사도

### ■ 1) 코사인 유사도 (Cosine Similarity)

- 두 벡터가 얼마나 같은 "방향"을 가리키는지를 측정

$$\cos\_sim(A, B) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^n a_i b_i}{\sqrt{\sum_{i=1}^n a_i^2} \sqrt{\sum_{i=1}^n b_i^2}}$$

- $A \cdot B = \sum_{i=1}^n a_i b_i \rightarrow$  두 벡터의 **내적**
- $\|A\| = \sqrt{\sum_{i=1}^n a_i^2}, \|B\| = \sqrt{\sum_{i=1}^n b_i^2} \rightarrow$  각 벡터의 **크기**
- 결과 값 범위:  $-1 \leq \cos\_sim(A, B) \leq 1$ 
  - 1  $\rightarrow$  완전히 같은 방향
  - 0  $\rightarrow$  직각, 관계 없음
  - -1  $\rightarrow$  완전히 반대 방향

## ❖ 단어 간 유사도

### ■ 2) 유클리드 거리 (Euclidean Distance)

- 두 벡터 간의 실제 "거리"를 계산
  - 벡터  $A=(a_1, a_2, \dots, a_n)$ ,  $B=(b_1, b_2, \dots, b_n)$  일 때,

$$d(A, B) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \dots + (a_n - b_n)^2} = \sqrt{\sum_{i=1}^n (a_i - b_i)^2}$$

- $(a_i - b_i)^2 \rightarrow$  각 차원의 좌표 차이의 제곱
- $\sum \rightarrow$  모든 차원의 차이를 합산
- $\sqrt{\cdot} \rightarrow$  최종 거리 계산

### 3. 단어 임베딩을 위한 분포가설/분산 표현

#### ❖ 단어 간 유사도

기준	코사인 유사도	유클리드 거리
의미	두 벡터의 방향이 얼마나 비슷한지	두 벡터 간 실제 거리
범위	-1 ~ 1	0 ~ ∞
장점	크기(scale)에 영향을 덜 받음	직관적, 이해하기 쉬움
활용	단어 임베딩 비교, 문서 유사도	군집 분석, KNN 등 거리 기반 알고리즘

### 3. 단어 임베딩을 위한 분포가설/분산 표현

#### ❖ 분산 표현 기반 임베딩 모델

구분	모델	벡터 특성	의미 반영	문맥 반영	장점	단점
초기 임베딩	<b>Word2Vec</b>	밀집, 저차원	O	X	단어 의미 학습	문맥 무시
통계 기반	<b>GloVe</b>	밀집, 저차원	O	X	코퍼스 전체 의미 반영	문맥 무시
문맥 기반	<b>BERT, GPT</b>	밀집, 저차원	O	O	문맥 반영, 다양한 NLP task 가능	연산량 많음

## ❖ Word2Vec (2013, Google-Mikolov et al.)

### ■ 예측 기반 분산 표현 모델

- 분포 가설(Distributional Hypothesis)에 기반. 단어를 예측하는 신경망 기반 모델, 학습을 통해 단어를 밀집(dense) 벡터로 표현
- 단어들을 "의미 공간"에 배치하여 단어들 간의 거리와 방향에 단어 의미 표현
- 특정 문장에서 자주 함께 등장하는 단어는 벡터 공간에서 가까운 위치를 갖게 됨
- CBOW: 주변 단어(context)로 중심 단어 예측.
- Skip-gram: 중심 단어로 주변 단어 예측.

### ■ 장점:

- 벡터 연산 가능 ( $\text{king} - \text{man} + \text{woman} \approx \text{queen}$ ). <https://word2vec.kr/search/>
- 단어 간 유사도 계산에 효과적

### ■ 한계:

- 단어 하나당 하나의 벡터(고정 벡터)만 존재해서, 다의어(예: "bank")를 구분하지 못함

### ■ Python 라이브러리

- `from gensim.models import Word2Vec`

## ❖ GloVe (2014, Stanford-Pennington et al.)

- 전역 통계 기반 분산 표현 모델. 전체 코퍼스의 단어-단어 동시 출현 행렬(co-occurrence matrix)을 기반으로 임베딩 학습
- 전역적(global) 통계 정보 + 지역적(local) 문맥 정보를 모두 반영
  - 전역적(global) 통계 정보:
    - 코퍼스 전체에서 단어가 얼마나 자주 함께 등장하는지 파악
    - 단어 간 전체적인 의미적 관계를 학습
  - 지역적(local) 문맥 정보:
    - Word2Vec과 마찬가지로, 특정 문맥에서의 단어 출현 정보도 반영
- 장점:
  - 전역적 통계를 고려해 Word2Vec보다 안정적인 표현 가능
- 한계:
  - 여전히 고정 벡터를 사용. 다의어 처리 불가
- *Python 라이브러리*

```
import gensim.downloader as api  
glove = api.load("glove-wiki-gigaword-50")
```

## ❖ FastText (2016, Facebook)

- 단어를 문자 n-gram으로 분해하여 벡터 학습
  - 희귀 단어, 신조어도 잘 처리. 한국어 같은 형태소 기반 언어에 강점
- 장점:
  - 어휘에 없는 단어도 일반화 가능
- 한계:
  - 여전히 문맥에 따른 의미 변화는 반영 못함.
- *Python 라이브러리*
  - `from gensim.models import FastText`

## ❖ ELMo (2018, Peters et al.)

- LSTM 기반의 양방향 언어모델. 문맥마다 다른 단어 벡터를 생성 (동적 벡터)
- 장점:
  - 다의어 문제 해결. 문맥에 따라 의미 달라짐을 반영
- 한계:
  - LSTM 구조의 한계(긴 문맥 처리 어려움), 학습/추론이 느림

### ■ Python 라이브러리

```
import tensorflow_hub as hub
```

```
# ELMo 모델 로드
```

```
elmo = hub.load("https://tfhub.dev/google/elmo/3")
```

## ❖ BERT (2019, Google-Devlin et al.)

- Transformer 기반 양방향 사전학습 모델
- 학습 방식: Masked Language Modeling (MLM), Next Sentence Prediction (NSP)
- 장점:
  - 문맥 기반 임베딩 생성.
  - 다양한 NLP 태스크에서 뛰어난 성능
- 한계:
  - 대규모 데이터와 연산 자원 필요. 학습·추론 비용이 큼
- *Python 라이브러리*
  - from sentence\_transformers import **SentenceTransformer**  
# BERT 모델 로드  
bert\_model = SentenceTransformer('all-MiniLM-L6-v2')



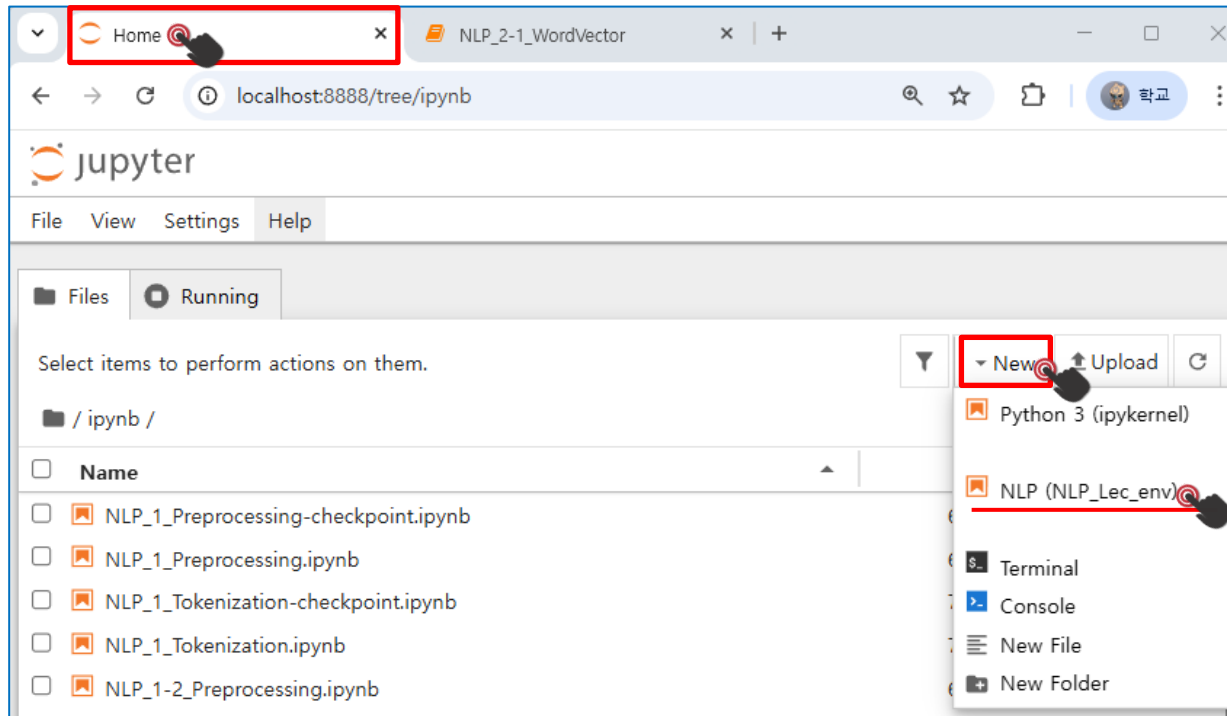
## ❖ GPT 계열 (2018~)

- Transformer 기반 단방향(왼쪽→오른쪽) 언어모델
- 대규모 데이터 학습, 텍스트 생성에 최적화된 모델
- 장점:
  - 자연스러운 텍스트 생성, 대화형 모델로 확장 가능
- 한계:
  - 단방향성 → 특정 태스크에 불리.
  - 모델 크기 증가로 비용 및 에너지 문제 발생

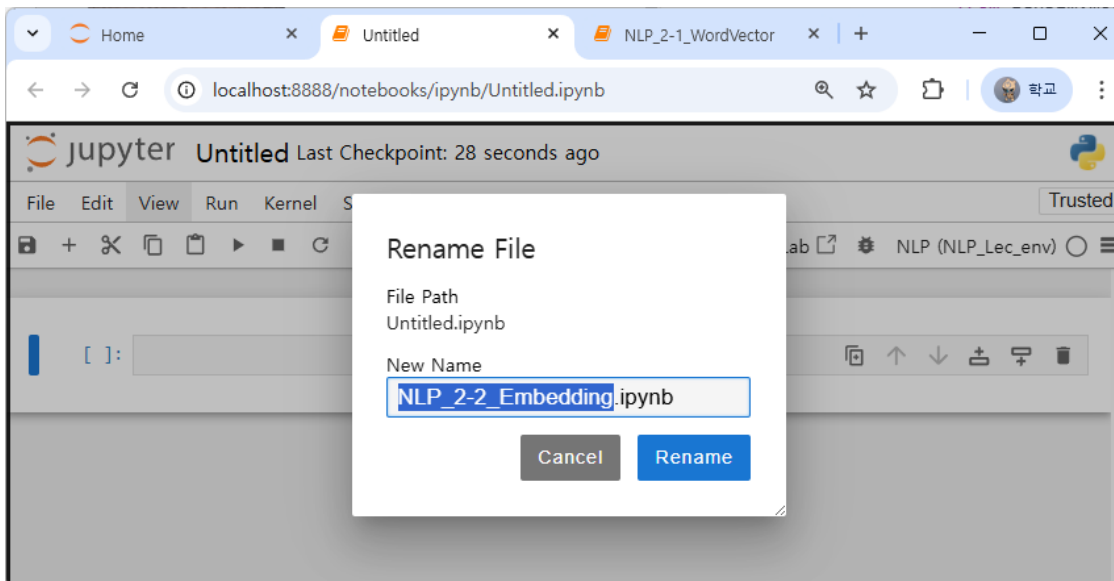
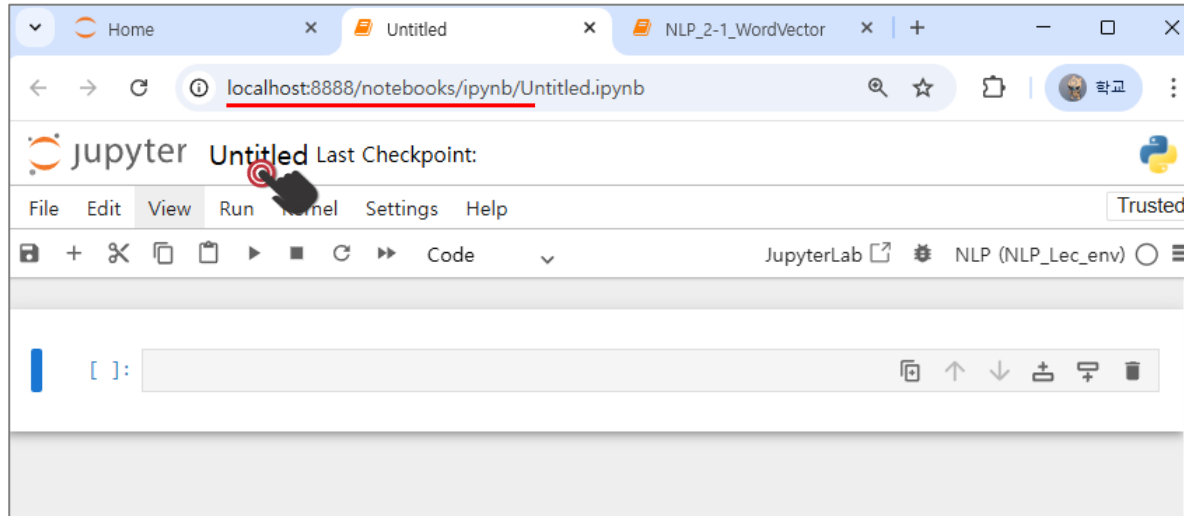


# 실습 2 준비

## ■ 1) 새 파일 생성



## 2) 파일 이름 변경



# 실습 2 준비



## ■ 필요한 패키지 설치

```
Anaconda Powershell Prompt
(NLP_Lec_env) (base) PS C:\Users\User\NLP_PJT>
(NLP_Lec_env) (base) PS C:\Users\User\NLP_PJT>
(NLP_Lec_env) (base) PS C:\Users\User\NLP_PJT> pip install sentence-transformers matplotlib seaborn
```



# 실습 2



## Word2Vec

- tokenized\_sentences
  - 학습할 토큰화된 문장 리스트
- vector\_size
  - 각 단어를 표현할 벡터의 차원 수
- window
  - 컨텍스트 윈도우 크기
  - 주변 단어 몇 개를 고려할지 설정
- min\_count
  - 최소 단어 빈도 수 설정
- workers
  - 학습에 사용할 병렬 처리 스레드 수

## 1. Word2Vec

```
[1]: import matplotlib.pyplot as plt
import seaborn as sns

[2]: sentences = [
    "I study natural language processing",
    "I study deep learning",
    "Language models capture meaning",
    "I went to the bank to deposit money",
    "The river bank was full of beautiful flowers"
]

[3]: tokenized_sentences = [s.lower().split() for s in sentences]

[4]: from gensim.models import Word2Vec

[5]: w2v_model = Word2Vec(tokenized_sentences, vector_size=50, window=5, min_count=1, workers=4)

[6]: # 단어 벡터 추출
words = ["language", "models", "bank", "money", "flowers"]
w2v_vectors = [w2v_model.wv[w] for w in words]

[7]: w2v_vectors

[7]: [array([-0.01723938,  0.00733148,  0.01037977,  0.01148388,  0.01493384,
          -0.01233535,  0.00221123,  0.01209456, -0.0056801 , -0.01234705,
          -0.00082045, -0.0167379 , -0.01120002,  0.01420908,  0.00670508,
           0.01445134,  0.01360049,  0.01506148, -0.00757831, -0.00112361,
           0.00469675, -0.00903806,  0.01677746, -0.01971633,  0.01352928,
           0.00582883, -0.00986566,  0.00879638, -0.00347915,  0.01342277,
           0.0199297 , -0.00872489, -0.00119868, -0.01139127,  0.00770164,
           0.00557325,  0.01378215,  0.01220219,  0.01907699,  0.01854683,
           0.01579614, -0.01397901, -0.01831173, -0.00071151, -0.00619968,
           0.01578863,  0.01187715, -0.00309133,  0.00302193,  0.00358008],
          dtype=float32),
      array([ 0.0054923 , -0.01672347,  0.01571267,  0.01706966, -0.01917073,
           0.00489962,  0.01981336, -0.0153274 , -0.01393429, -0.01546833,
           0.01678944, -0.00136178,  0.01828859, -0.01631665,  0.00748822,
           0.00527341,  0.00148759,  0.00465837, -0.01493756, -0.01871562,
           0.00470773,  0.01229179,  0.01597698,  0.0114763 , -0.00155723,
           0.01660943, -0.01867347,  0.00680943,  0.00053512,  0.00771154,
           0.01476474, -0.01344972,  0.01116867, -0.01904787, -0.00160761,
```

# 실습 2



## ■ TSNE

- 고차원 데이터를 저차원(보통 2차원, 3차원)으로 변환하여 시각화할 때 주로 사용
- 데이터 포인트 간 유사도를 보존하면서 차원 축소
- n\_components=2
  - 축소할 차원의 수 설정.
- perplexity
  - SNE의 중요한 하이퍼파라미터 중 하나
  - 주변 이웃의 수를 고려하는 정도를 결정
  - 전체 샘플의 개수를 초과할 수 없음!!
  - 일반적으로 5~50 범위의 값 사용
  - 값이 너무 작으면 지역 구조(local structure)에만 집중, 너무 크면 글로벌 구조(global structure)에 집중

## 2D 시각화를 위한 차원 축소 (t-SNE)

```
[8]: from sklearn.manifold import TSNE

[9]: import numpy as np

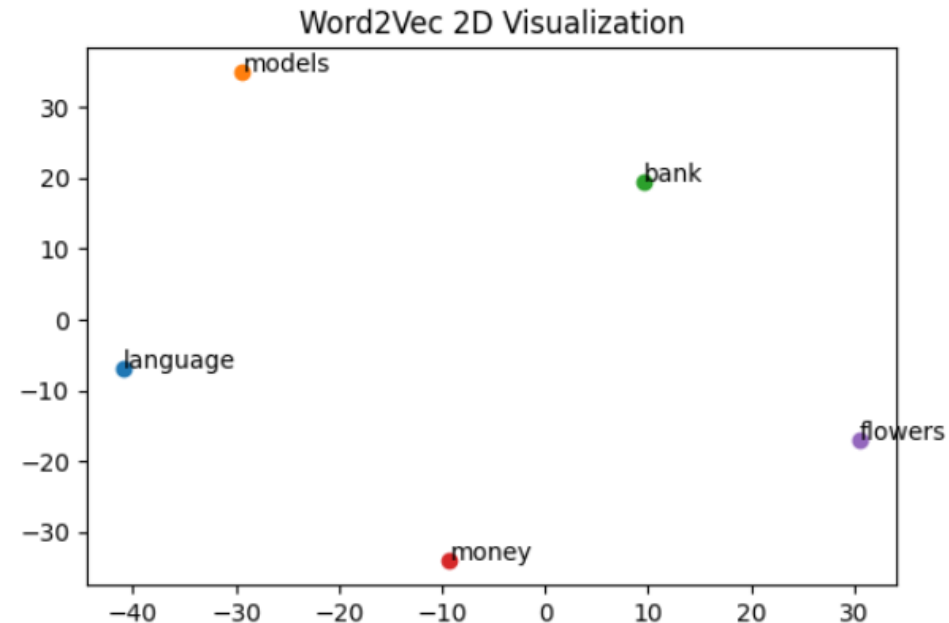
[10]: # Word2Vec 시각화 tsne 입력 형태인 array로 변형
X_w2v_vectors = np.array(w2v_vectors) # (n_samples, n_features)
n_samples = X_w2v_vectors.shape[0]
perplexity = max(1, min(30, n_samples - 1))

[11]: # Word2Vec 시각화 tsne 학습
tsne = TSNE(n_components=2, random_state=42, perplexity=perplexity)
w2v_2d = tsne.fit_transform(X_w2v_vectors)

[12]: plt.figure(figsize=(6,4))

for i, word in enumerate(words):
    plt.scatter(w2v_2d[i,0], w2v_2d[i,1])
    plt.text(w2v_2d[i,0]+0.01, w2v_2d[i,1]+0.01, word)

plt.title("Word2Vec 2D Visualization")
plt.show()
```



# 실습 2

## 2. Glove

```
[13]: import gensim.downloader as api

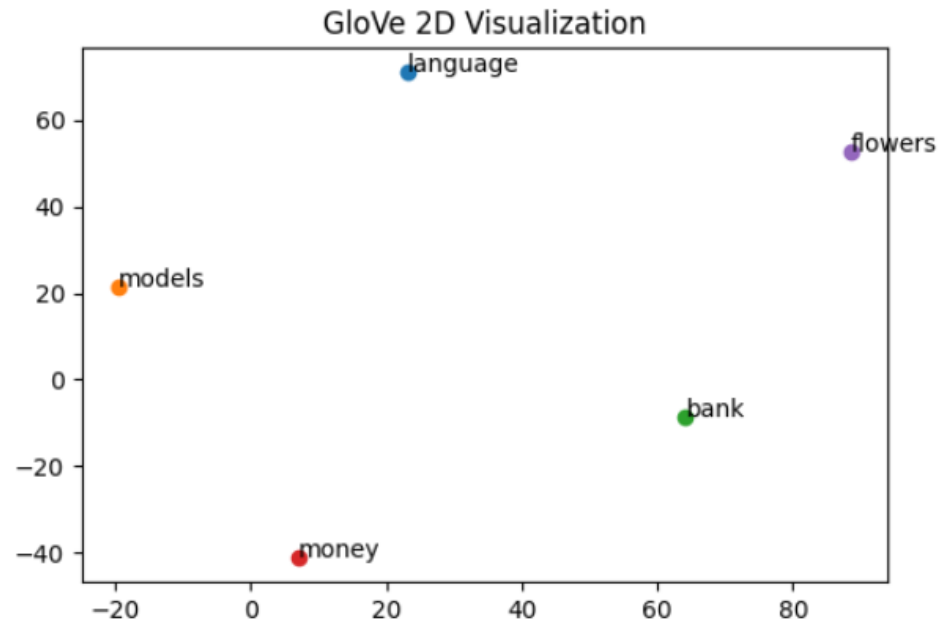
[14]: glove_model = api.load("glove-wiki-gigaword-50")

[15]: glove_vectors = [glove_model[w] for w in words]

[16]: # 시각화 tsne 입력 형태인 array로 변형
X_glove_vectors = np.array(glove_vectors) # (n_samples, n_features)
n_samples = X_glove_vectors.shape[0]
perplexity = max(1, min(30, n_samples - 1))

[17]: # GloVe 시각화
glove_2d = TSNE(n_components=2, random_state=42, perplexity=perplexity).fit_transform(X_glove_vectors)

plt.figure(figsize=(6,4))
for i, word in enumerate(words):
    plt.scatter(glove_2d[i,0], glove_2d[i,1])
    plt.text(glove_2d[i,0]+0.01, glove_2d[i,1]+0.01, word)
plt.title("GloVe 2D Visualization")
plt.show()
```



## 실습 2

### 3. BERT

```
[18]: from sentence_transformers import SentenceTransformer
import numpy as np

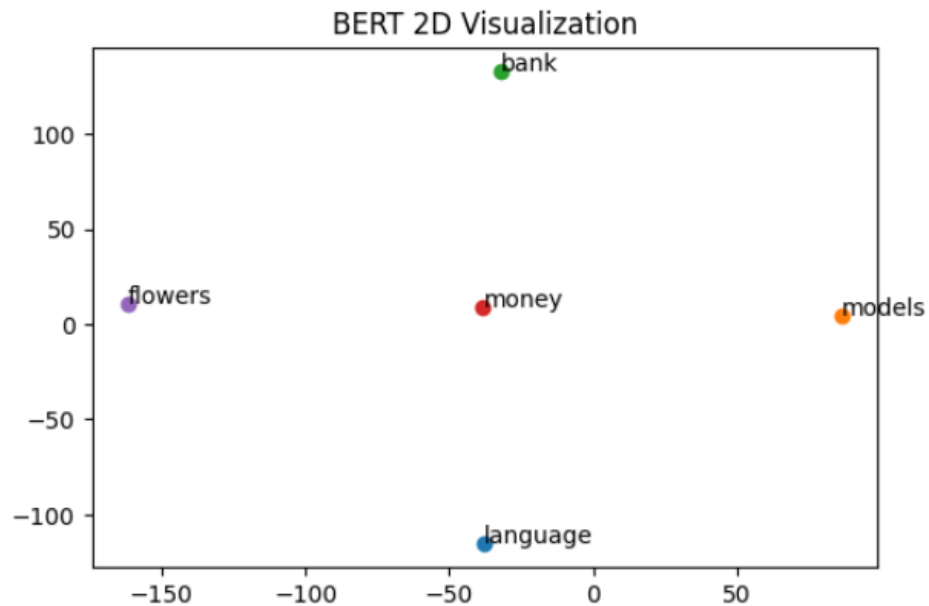
[19]: bert_model = SentenceTransformer('all-MiniLM-L6-v2')

[20]: # 단어 수준 확인을 위해 문장 단위로 추출 후 첫 단어 임베딩 사용
bert_embeddings = bert_model.encode(sentences)

[21]: # 시각화 tsne 입력 형태인 array로 변형
X_bert_vectors = np.array(bert_embeddings) # (n_samples, n_features)
n_samples = X_bert_vectors.shape[0]
perplexity = max(1, min(30, n_samples - 1))

[22]: bert_2d = TSNE(n_components=2, random_state=42, perplexity=perplexity).fit_transform(X_bert_vectors)

plt.figure(figsize=(6,4))
bert_words = ["language", "models", "bank", "money", "flowers"]
for i, word in enumerate(bert_words):
    plt.scatter(bert_2d[i,0], bert_2d[i,1])
    plt.text(bert_2d[i,0]+0.01, bert_2d[i,1]+0.01, word)
plt.title("BERT 2D Visualization")
plt.show()
```



# 단어 임베딩 모델 정리

방법	특징	장점	단점(한계)
원-핫 인코딩 (One-Hot Encoding)	<ul style="list-style-type: none"> <li>단어를 고정된 크기의 벡터로 표현</li> <li>해당 단어 위치만 1, 나머지는 0</li> </ul>	<ul style="list-style-type: none"> <li>단순하고 구현이 쉬움</li> <li>단어를 구분 가능</li> </ul>	<ul style="list-style-type: none"> <li>차원이 매우 큼(어휘 크기 = 벡터 차원) -&gt; 희소(sparse) 벡터</li> <li>단어 의미/유사성 반영 불가</li> </ul>
카운트 기반 표현 (Bag-of-Words, TF-IDF)	<ul style="list-style-type: none"> <li>문서에서 단어 출현 횟수(빈도)를 사용</li> <li>TF-IDF는 단어의 중요도를 가중치로 반영</li> </ul>	<ul style="list-style-type: none"> <li>구현이 단순</li> <li>정보 검색(IR)에서 유용</li> </ul>	<ul style="list-style-type: none"> <li>단어 순서, 문맥 정보 손실</li> <li>차원이 큼</li> <li>단어 의미 유사성 반영 불가</li> </ul>
Word2Vec (Mikolov, 2013)	<ul style="list-style-type: none"> <li>예측 기반 모델 (CBOW, Skip-gram)</li> <li>분포 가설 기반 단어 임베딩 학습</li> </ul>	<ul style="list-style-type: none"> <li>밀집(dense) 벡터</li> <li>의미적 유사성 반영</li> <li>벡터 연산 가능</li> </ul>	<ul style="list-style-type: none"> <li>단어당 하나의 벡터만 존재 (다의어 처리 불가)</li> <li>문맥에 따른 의미 변화 반영 불가</li> </ul>
GloVe (Pennington, 2014)	<ul style="list-style-type: none"> <li>단어 동시출현 행렬 기반 + 통계적 최적화</li> <li>전역적 통계 정보 반영</li> </ul>	<ul style="list-style-type: none"> <li>전역적 의미 반영</li> <li>계산 효율성 높음</li> </ul>	<ul style="list-style-type: none"> <li>Word2Vec과 같은 한계 (문맥 비반영, 다의어 처리 불가)</li> <li>대규모 행렬 계산 필요</li> </ul>
FastText (Facebook, 2016)	<ul style="list-style-type: none"> <li>단어를 문자 n-gram 단위로 분해하여 학습</li> <li>희귀 단어/신조어도 처리 가능</li> </ul>	<ul style="list-style-type: none"> <li>형태소 기반 언어(한국어 등)에 강점</li> <li>희귀 단어 일반화 가능</li> </ul>	<ul style="list-style-type: none"> <li>여전히 단어별 고정 벡터</li> <li>문맥 의존 의미 반영 불가</li> </ul>
ELMo (Peters, 2018)	<ul style="list-style-type: none"> <li>LSTM 기반 양방향 언어모델</li> <li>단어의 문맥별 동적 표현 생성</li> </ul>	<ul style="list-style-type: none"> <li>문맥 의존적 단어 임베딩</li> <li>다의어 문제 해결</li> </ul>	<ul style="list-style-type: none"> <li>학습/추론 속도가 느림</li> <li>LSTM 기반 → 긴 문맥 처리 한계</li> </ul>
BERT (Devlin, 2019)	<ul style="list-style-type: none"> <li>Transformer 기반 양방향 언어모델</li> <li>Masked Language Modeling(MLM) 학습</li> </ul>	<ul style="list-style-type: none"> <li>문맥 기반 임베딩 생성</li> <li>문장/단어 수준에서 의미 풍부</li> <li>다양한 NLP 태스크에서 고성능</li> </ul>	<ul style="list-style-type: none"> <li>사전학습/추론 비용 큼</li> <li>고비용 GPU 필요</li> <li>대규모 데이터 의존</li> </ul>
GPT 계열 (2018~)	<ul style="list-style-type: none"> <li>Transformer 기반 단방향(왼→오른쪽) 언어모델</li> <li>대규모 데이터로 사전학습</li> </ul>	<ul style="list-style-type: none"> <li>문맥 기반 언어 생성에 최적화</li> <li>자연스러운 텍스트 생성</li> <li>대화형 응용 가능</li> </ul>	<ul style="list-style-type: none"> <li>단방향성으로 특정 태스크에 불리</li> <li>매우 큰 모델 → 학습 비용, 에너지 소모</li> </ul>
최신 임베딩 (e.g., BERT 변형, GPT, LLaMA, Mistral 등)	<ul style="list-style-type: none"> <li>대규모 파라미터 기반 멀티모달 확장</li> <li>Zero-shot, Few-shot 학습 가능</li> </ul>	<ul style="list-style-type: none"> <li>범용적 활용 가능</li> <li>멀티태스크/멀티모달 지원</li> </ul>	<ul style="list-style-type: none"> <li>모델 크기 증가 → 비용 문제</li> <li>블랙박스적 해석 어려움</li> </ul>

# 단어 임베딩 모델 활용

기법	서비스 활용 분야	사례	특징
Bag-of-Words (BoW)	문서 분류, 스팸 탐지	<ul style="list-style-type: none"> <li>- 이메일 스팸 필터링 (단순히 특정 단어 출현 빈도로 분류)</li> <li>- 영화 리뷰 감성 분류(긍/부정)</li> </ul>	<ul style="list-style-type: none"> <li>- 단순하면서도 빠르게 구현 가능</li> <li>- 단어 순서, 문맥 정보는 반영하지 못함</li> </ul>
TF-IDF	검색 엔진, 추천 시스템	<ul style="list-style-type: none"> <li>- Google 초창기 검색 랭킹 알고리즘(문서 내 단어의 중요도를 반영)</li> <li>- 뉴스 기사 추천(주요 키워드 기반 유사 기사 추천)</li> <li>- 학술 논문 검색(논문 내 중요한 키워드 강조)</li> </ul>	<ul style="list-style-type: none"> <li>- 흔한 단어는 가중치를 낮추고, 드문 단어는 가중치를 높임</li> <li>- 여전히 단어 순서와 문맥은 반영 못함</li> </ul>
Word2Vec	챗봇, 추천 시스템, 번역	<ul style="list-style-type: none"> <li>- 구글 뉴스 단어 임베딩 (단어 간 유사도 학습)</li> <li>- Netflix 영화 추천 (영화 설명 텍스트에서 의미 벡터 추출, 비슷한 영화 추천)</li> <li>- Siri, Alexa 같은 음성비서 (사용자 질문 이해 및 의미 파악)</li> </ul>	<ul style="list-style-type: none"> <li>- 단어 간 의미적 유사성 반영</li> <li>- 학습 데이터의 품질에 성능 크게 의존</li> </ul>
GloVe (Global Vectors)	지식 그래프, 검색/추천, NLP 응용	<ul style="list-style-type: none"> <li>- Wikipedia + Common Crawl 대규모 코퍼스 학습으로 사전 학습된 임베딩 제공</li> <li>- IBM Watson에서 의미 기반 질의응답</li> <li>- Semantic Search (예: 유사 문서 검색, FAQ 매칭)</li> </ul>	<ul style="list-style-type: none"> <li>- 전역 통계정보(동시 출현 빈도) 반영</li> <li>- Word2Vec보다 더 넓은 문맥 통계 고려</li> </ul>



## ❖ 압축하여, 구글 클래스룸에 제출.

압축파일 이름: NLP\_수업일\_자기이름

- NLP\_1-2\_Preprocessing
- NLP\_2-1\_WordVector
- NLP\_2-2\_Embedding

