딥러닝을 이용한 자연어처리 입문

Ch05. 벡터의 유사도

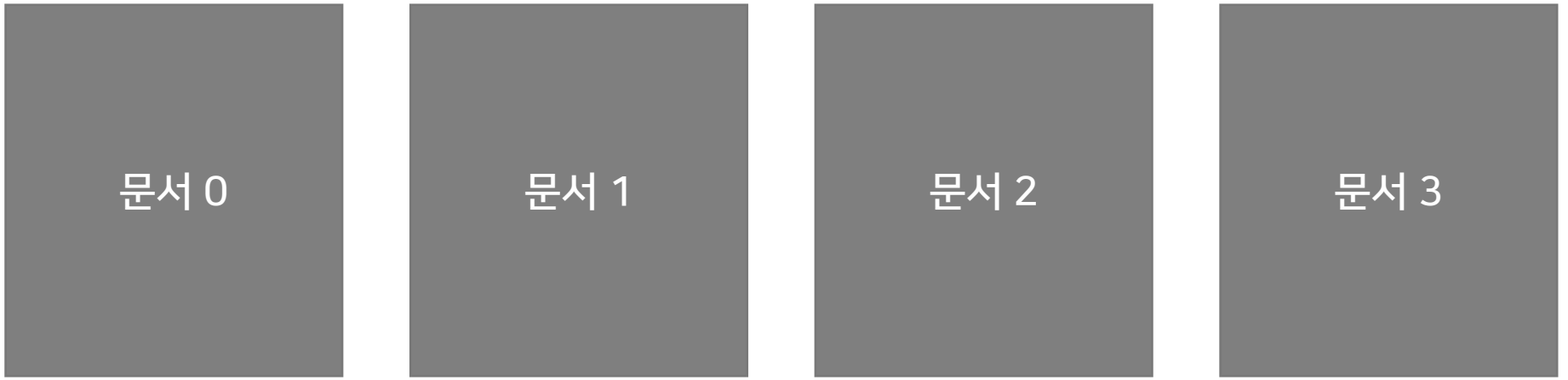
집현전 초급반 5조 : 박서영, 지현구, 조성철(팀장)

Contents

1. Introduction

Introduction

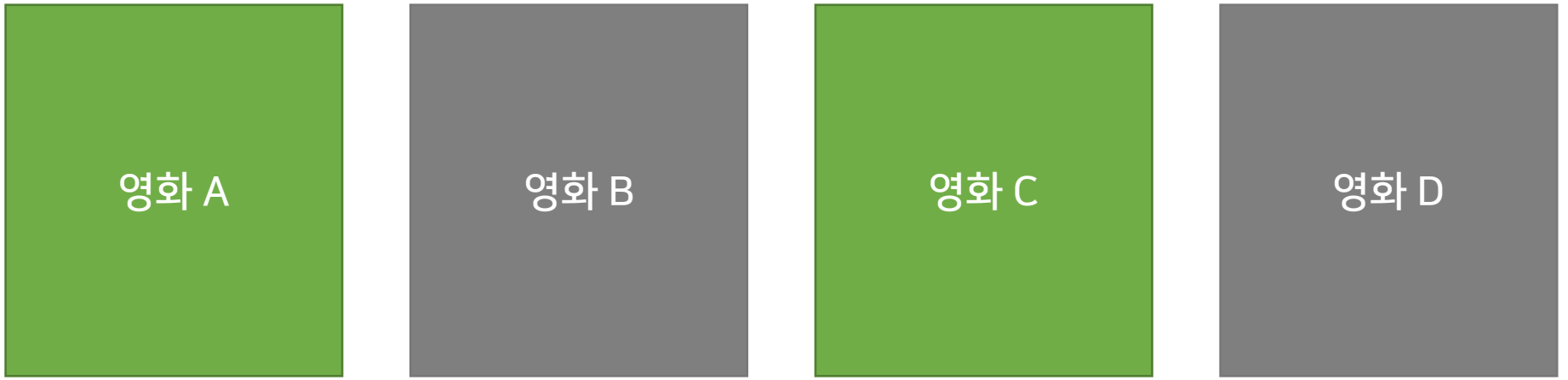
- **문서의 유사도 계산은 왜 하는가?**



어떤 문서가 유사한가?

Introduction

- **문서의 유사도 계산은 왜 하는가?**



어떤 영화가 유사한가?

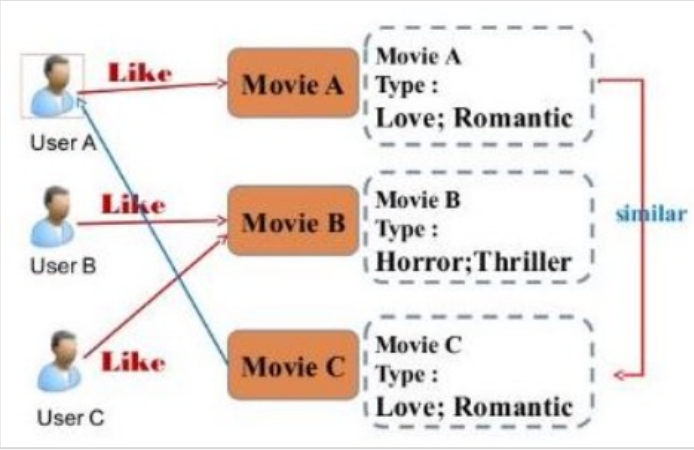
영화 A를 시청한 사람에게 유사한 영화 C를 추천해주는 추천시스템

Introduction

왜 추천 시스템이 중요한가? - 실제 활용사례

콘텐츠 기반 필터링(content based filtering)

- 사용자가 특정 아이템을 선호하는 경우 그 아이템과 비슷한 콘텐츠를 가진 다른 아이템을 추천
- 예) 사용자 A가 ItemA에 굉장히 높은 평점을 주었는데 그 Item이 액션 영화이며 '이수진 ' 이라는 감독이었으면 '이수진 ' 감독의 다른 액션 영화를 추천
- 과거에 자주 사용했던 방식



협업 필터링(collaborative filtering)

- 넷플릭스(netflix) 사례 이후 협업 필터링을 많이 사용
- 기존 사용자의 행동 정보를 분석해 해당 사용자와 비슷한 성향의 사용자들이 기존에 좋아했던 항목을 추천
- 즉, 사용자의 행동 기록을 이용
- 예) 드라마 <하우스 오브 카드>를 시청한 시청자가 <홈랜드>를 시청한 경우가 많으면 <하우스 오브 카드>를 시청한 사람에게 홈랜드를 추천

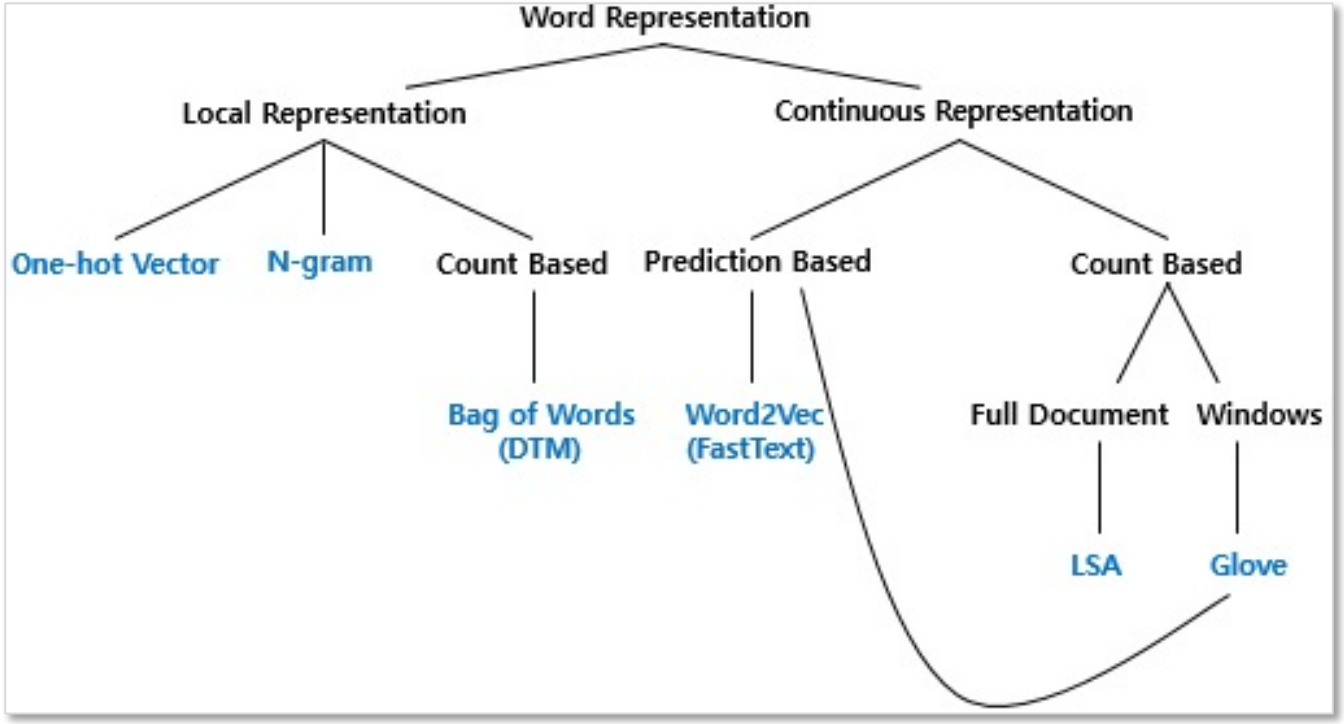
2. 문서간 유사도 계산

1) 코사인 유사도(Cosine Similarity)

2) 유사도를 이용한 추천시스템 구현

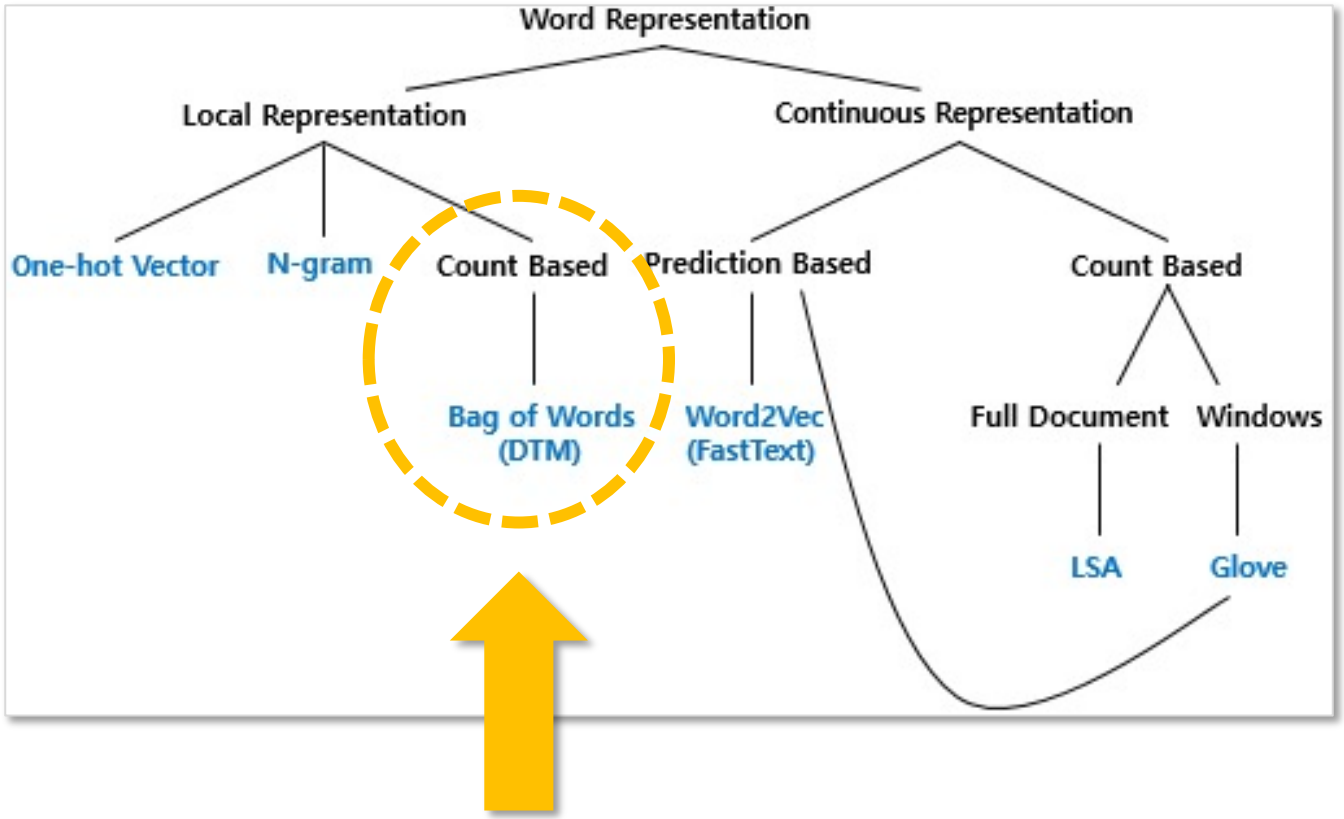
Review

- 단어의 표현방법



Review

- 단어의 표현방법



Review

단어의 표현방법

BoW

```
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}
```

출현 빈도를 통한 텍스트데이터 수치화 기법
1. 각 단어에 고유한 정수 인덱스 부여
2. 인덱스 위치에 단어 토큰의 등장 횟수를 기록한 벡터

문서단어행렬 DTM

서로 다른 문서들의 Bow들을 결합

	과일이	길고	노란	먹고	바나나	사과	싶은	저는	좋아요
0	0	0	0	1	0	1	1	0	0
1	0	0	0	1	1	0	1	0	0
2	0	1	1	0	2	0	0	0	0
3	1	0	0	0	0	0	0	1	1

TF

TF-IDF

DTM에 불용어&중요단어 가중치를 부여

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

	IDF
과일이	0.693147
길고	0.693147
노란	0.693147
먹고	0.287682
바나나	0.287682
사과	0.693147
싶은	0.287682
저는	0.693147
좋아요	0.693147

IDF

	과일이	길고	노란	먹고	바나나	사과	싶은	저는	좋아요
0	0.000000	0.000000	0.000000	0.287682	0.000000	0.693147	0.287682	0.000000	0.000000
1	0.000000	0.000000	0.000000	0.287682	0.287682	0.000000	0.287682	0.000000	0.000000
2	0.000000	0.693147	0.693147	0.000000	0.575364	0.000000	0.000000	0.000000	0.000000
3	0.693147	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.693147	0.693147

TF-IDF

Review

단어의 표현방법

BoW

```
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}
```

출현 빈도를 통한 텍스트데이터 수치화 기법
1. 각 단어에 고유한 정수 인덱스 부여
2. 인덱스 위치에 단어 토큰의 등장 횟수를 기록한 벡터

문서단어행렬 DTM

서로 다른 문서들의 Bow들을 결합

	과일이	길고	노란	먹고	바나나	사과	싶은	저는	좋아요
0	0	0	0	1	0	1	1	0	0
1	0	0	0	1	1	0	1	0	0
2	0	1	1	0	2	0	0	0	0
3	1	0	0	0	0	0	0	1	1

TF

TF-IDF

DTM에 불용어&중요단어 가중치를 부여

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

	IDF
과일이	0.693147
길고	0.693147
노란	0.693147
먹고	0.287682
바나나	0.287682
사과	0.693147
싶은	0.287682
저는	0.693147
좋아요	0.693147

IDF

	과일이	길고	노란	먹고	바나나	사과	싶은	저는	좋아요
0	0.000000	0.000000	0.000000	0.287682	0.000000	0.693147	0.287682	0.000000	0.000000
1	0.000000	0.000000	0.000000	0.287682	0.287682	0.000000	0.287682	0.000000	0.000000
2	0.000000	0.693147	0.693147	0.000000	0.575364	0.000000	0.000000	0.000000	0.000000
3	0.693147	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.693147	0.693147

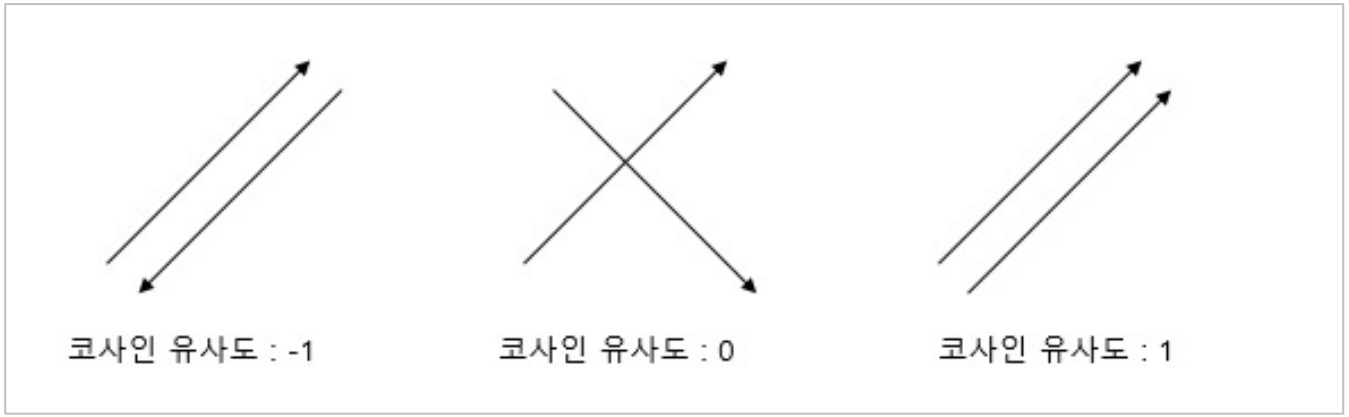
TF-IDF

문서번호

문서간 유사도 계산

코사인 유사도(Cosine Similarity)

- 두 벡터 간의 코사인 각도를 이용하여 구할 수 있는 두 벡터의 유사도
- -1 이상 1 이하
- 값이 1에 가까울수록 유사도가 높다고 판단
- 두 벡터가 가리키는 방향이 얼마나 유사한가를 의미



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

문서간 유사도 계산

■ 코사인 유사도(Cosine Similarity)

- 예시

문서 1 저는 사과 좋아요

문서 2 저는 바나나 좋아요

문서 3 저는 바나나 좋아요 저는 바나나 좋아요

-	바나나	사과	저는	좋아요
문서1	0	1	1	1
문서2	1	0	1	1
문서3	2	0	2	2

문서단어행렬 DTM

Numpy

```
from numpy import dot
from numpy.linalg import norm
import numpy as np
def cos_sim(A, B):
       return dot(A, B)/(norm(A)*norm(B))
```

```
doc1=np.array([0,1,1,1])
doc2=np.array([1,0,1,1])
doc3=np.array([2,0,2,2])
```

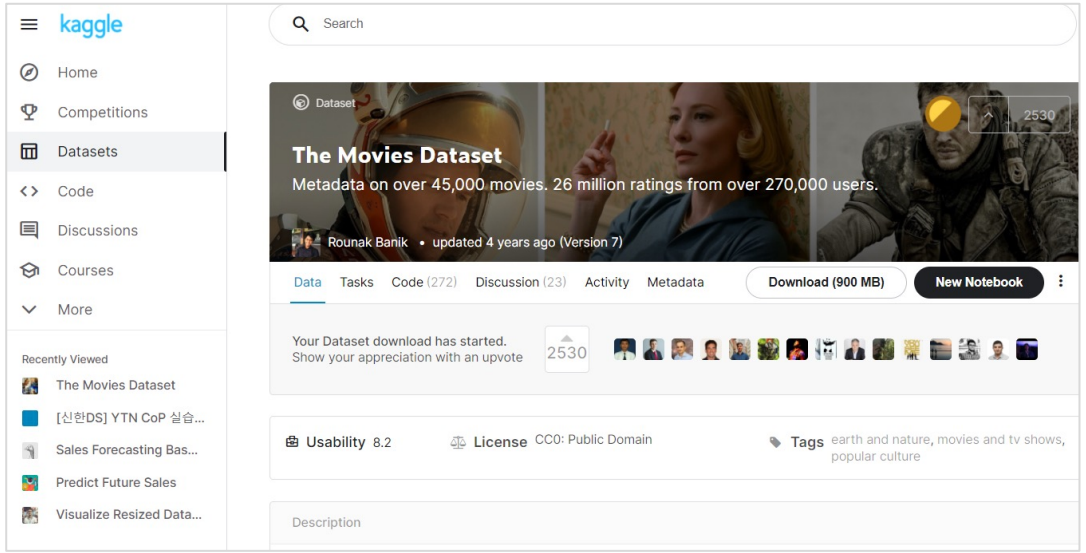
```
print(cos_sim(doc1, doc2)) #문서1과 문서2의 코사인 유사도
print(cos_sim(doc1, doc3)) #문서1과 문서3의 코사인 유사도
print(cos_sim(doc2, doc3)) #문서2과 문서3의 코사인 유사도
```

```
0.6666666666666667
0.6666666666666667
1.0000000000000002
```

문서간 유사도 계산

유사도를 이용한 추천시스템 구현

- 실습 데이터셋
  - Kaggle - The Movie Dataset (https://www.kaggle.com/rounakbanik/the-movies-dataset?select=movies_metadata.csv)
  - 영화 제목('title')과 줄거리('overview') 컬럼 활용



	adult	belongs_to_collection	budget	genres	homepage	id	imdb_id	original_language	original_title	overview	...	release_d
0	False	{'id': 10194, 'name': 'Toy Story Collection', ...	30000000	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...	http://toystory.disney.com/toy-story	862	tt0114709	en	Toy Story	Led by Woody, Andy's toys live happily in his ...	...	1995-10
1	False	NaN	65000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...	NaN	8844	tt0113497	en	Jumanji	When siblings Judy and Peter discover an encha...	...	1995-12
2	False	{'id': 119050, 'name': 'Grumpy Old Men Collect...	0	[{'id': 10749, 'name': 'Romance'}, {'id': 35, ...	NaN	15602	tt0113228	en	Grumpier Old Men	A family wedding reignites the ancient feud be...	...	1995-12

	title	overview
0	Toy Story	Led by Woody, Andy's toys live happily in his ...
1	Jumanji	When siblings Judy and Peter discover an encha...
2	Grumpier Old Men	A family wedding reignites the ancient feud be...
3	Waiting to Exhale	Cheated on, mistreated and stepped on, the wom...
4	Father of the Bride Part II	Just when George Banks has recovered from his ...

문서간 유사도 계산

■ 유사도를 이용한 추천시스템 구현

- 실습(1/3)
  - 데이터 개수 4.5만->2만
  - Tf-idf 위한 결측치 처리
  - Overview에 대해서 tf-idf 수행
  - 코사인 유사도를 사용해서 문서간 유사도 구하기

```
data = data.head(20000)
```

```
data['overview'] = data['overview'].fillna('')
```

```
tfidf = TfidfVectorizer(stop_words='english')
# overview에 대해서 tf-idf 수행
tfidf_matrix = tfidf.fit_transform(data['overview'])
print(tfidf_matrix.shape)
```

```
(20000, 47487)
```

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

문서간 유사도 계산

■ 유사도를 이용한 추천시스템 구현

- 실습(2/3)
  - 영화 타이틀과 인덱스를 가진 테이블 생성
  - 선택한 영화에 대해서 코사인 유사도를 이용하여,
    가장 overview가 유사한 10개의 영화를 찾아내는 함수(def get_recommendations())

```
# 영화 타이틀과 인덱스를 가진 테이블 생성
indices = pd.Series(data.index, index=data['title']).drop_duplicates()
print(indices.head())
```

```
title
Toy Story                    0
Jumanji                      1
Grumpier Old Men             2
Waiting to Exhale            3
Father of the Bride Part II  4
dtype: int64
```

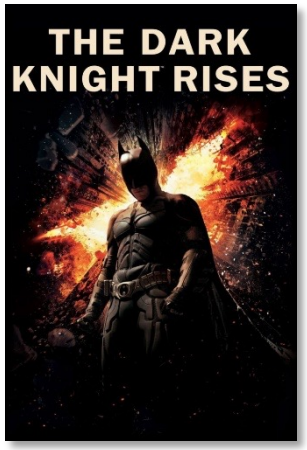
```
def get_recommendations(title, cosine_sim=cosine_sim):
    idx = indices[title]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:11]
    movie_indices = [i[0] for i in sim_scores]
    return data['title'].iloc[movie_indices]
```

- 선택한 영화 타이틀로부터 해당되는 인덱스(idx)
- 모든 영화에 대해서 해당 영화와의 유사도(sim_scores)
- 유사도에 따라 영화들을 정렬(sorted())
- 가장 유사한 10개의 영화(sim_scores[1:11])
- 영화 인덱스(movie_indices)
- 가장 유사한 10개의 영화 제목('title')을 리턴

문서간 유사도 계산

유사도를 이용한 추천시스템 구현

- 실습(3/3)
  - 영화 'The Dark Knight Rises'와 줄거리가 가장 유사한 영화는?



get_recommendations()

Top 1

Top 2

Top 3


```
get_recommendations('The Dark Knight Rises')
```

```
12481                         The Dark Knight
150                            Batman Forever
1328                           Batman Returns
15511                 Batman: Under the Red Hood
585                                     Batman
9230          Batman Beyond: Return of the Joker
18035                          Batman: Year One
19792    Batman: The Dark Knight Returns, Part 1
3095                Batman: Mask of the Phantasm
10122                           Batman Begins
Name: title, dtype: object
```

문서간 유사도 계산

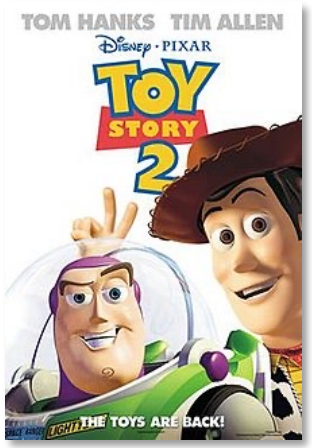
■ 유사도를 이용한 추천시스템 구현

- 실습(3/3)
  - 영화 'Toy Story'와 줄거리가 가장 유사한 영화는?

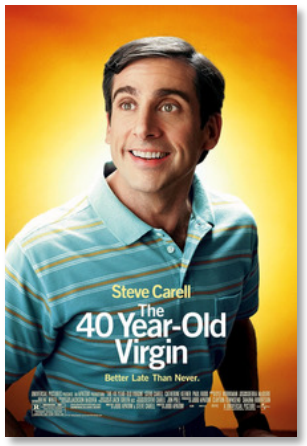
get_recommendations()

Top 1

Top 2



Top 3




```
get_recommendations('Toy Story')
```

```
15348                Toy Story 3
2997                 Toy Story 2
10301     The 40 Year Old Virgin
8327                   The Champ
1071       Rebel Without a Cause
11399     For Your Consideration
1932                   Condorman
3057            Man on the Moon
485                        Malice
11606               Factory Girl
Name: title, dtype: object
```

3. 여러가지 유사도 기법

1) 유클리드 거리(Euclidean distance)
2) 자카드 유사도(Jaccard similarity)
3) 그 외 유사도 측정 비교 및 활용

유클리드 거리(Euclidean distance)

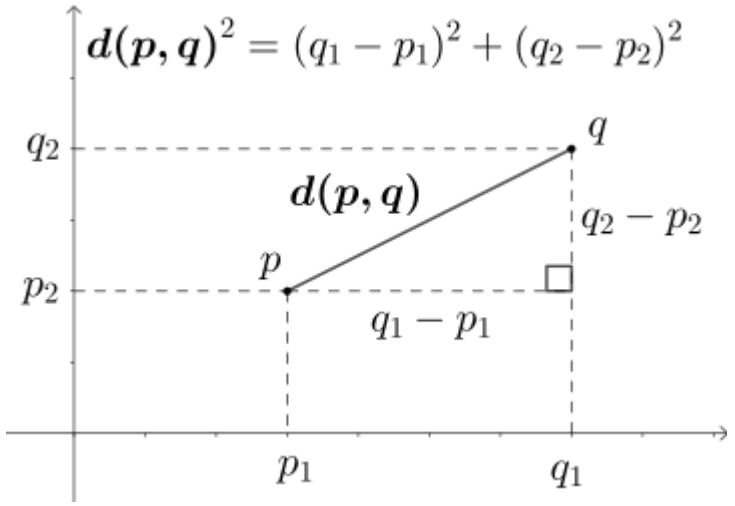
- **두 점 사이의 직선거리를 구하는 유클리드 거리 공식**

다차원 공간에서 두개의 점 p와 q가 각각 p=(p1,p2,p3,...,pn)과 q=(q1,q2,q3,...,qn)의 좌표를 가질 때 두 점 사이의 거리

$$\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ \dots \ + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

2차원 공간으로 가정하면 좌표 평면 상에서 두 점 p와 q사이의 직선 거리를 구하는 문제

** 피타고라스의 정리를 통해 두 점 사이의 거리를 구하는 것과 동일



유클리드 거리(Euclidean distance)

- **여러 문서에 대해서 유사도를 구하고자 유클리드 거리 공식을 사용**

단어의 개수가 4개이므로, 이는 4차원 공간에 문서1, 문서2, 문서3을 배치하는 것과 같음
이때 다음과 같은 문서Q에 대해서 문서1, 문서2, 문서3 중 가장 유사한 문서를 찾아내고자 함

-	바나나	사과	저는	좋아요
문서1	2	3	0	1
문서2	1	2	3	1
문서3	2	1	2	2

<문서단어행렬 DTM 예시>

문서Q 또한 다른 문서들처럼 4차원 공간에 배치시켰다는 관점에서 4차원 공간에서의 각각의 문서들과의 유클리드 거리를 구하면 됨

-	바나나	사과	저는	좋아요
문서Q	1	1	0	1

유클리드 거리(Euclidean distance)

- **파이썬 코드 구현**

  docQ에 대해서 doc1, doc2, doc3 중 가장 유사한 문서를 찾아내고자 함

  유클리드 거리 값이 가장 작다는 것은 문서상의 거리가 가장 가깝다는 의미

```
import numpy as np
def dist(x,y):
    return np.sqrt(np.sum((x-y)**2))

doc1 = np.array((2,3,0,1))
doc2 = np.array((1,2,3,1))
doc3 = np.array((2,1,2,2))
docQ = np.array((1,1,0,1))

print(dist(doc1,docQ))
print(dist(doc2,docQ))
print(dist(doc3,docQ))
```

* numpy import

* dist 함수 정의 $\sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\ \dots\ +(q_n-p_n)^2}=\sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}$

* 4차원 DTM(Document-Term Matrix, 문서 단어행렬) 생성

-	바나나	사과	저는	좋아요
문서1	2	3	0	1
문서2	1	2	3	1
문서3	2	1	2	2

-	바나나	사과	저는	좋아요
문서Q	1	1	0	1

* doc1, doc2, doc3 각각 docQ 와의 거리

```
2.23606797749979
3.1622776601683795
2.449489742783178
```

* doc1과 docQ 사이의 거리가 가장 가까움

➡ doc1이 docQ와 가장 유사

유클리드 거리(Euclidean distance)

■ 사이킷런 구현방식

* Scikit-learn은 Python 프로그래밍 언어를 위한 머신러닝 라이브러리

```
from sklearn.metrics.pairwise import euclidean_distances

euclidean_distances(tfidf_matrix[0:1], tfidf_matrix[1:2])
```

유사도 방식들은 모두 0과 1사이의 값을 가졌는데, 유클리디언 유사도는 1보다 큰 값이 나옴.

=> 두 점의 거리를 측정하는 것이므로 큰 값이 나올 수 있음.

```
import numpy as np

def l1_normalize(v):
        norm = np.sum(v)
    return v / norm


tfidf_norm_l1 = l1_normalize(tfidf_matrix)

euclidean_distances(tfidf_norm_l1[0:1], tfidf_norm_l1[1:2])
```

0과 1사이의 값으로 정규화를 하는 경우도 있음

정규화 방법 : 각 벡터안의 요소값을 모두 더한 것의 크기가 1이 되도록 벡터의 크기를 조절하는 방법
: 즉, 벡터의 모든 값을 더한 뒤, 이 값으로 각 벡터의 값을 나눈다.

출처 https://soyoung-new-challenge.tistory.com/34

맨해튼 거리/유사도(Manhattan distance/similarity)

- **맨해튼 거리(Manhattan distance, 혹은 택시 거리, L1 거리, 시가지 거리,Taxicab geometry)**

  19세기의 수학자 헤르만 민코프스키가 고안한 용어로, 보통 유클리드 기하학의 거리 공간을 좌표에 표시된 두 점 사이의 거리(절댓값)의 차이에 따른 새로운 거리 공간으로 대신

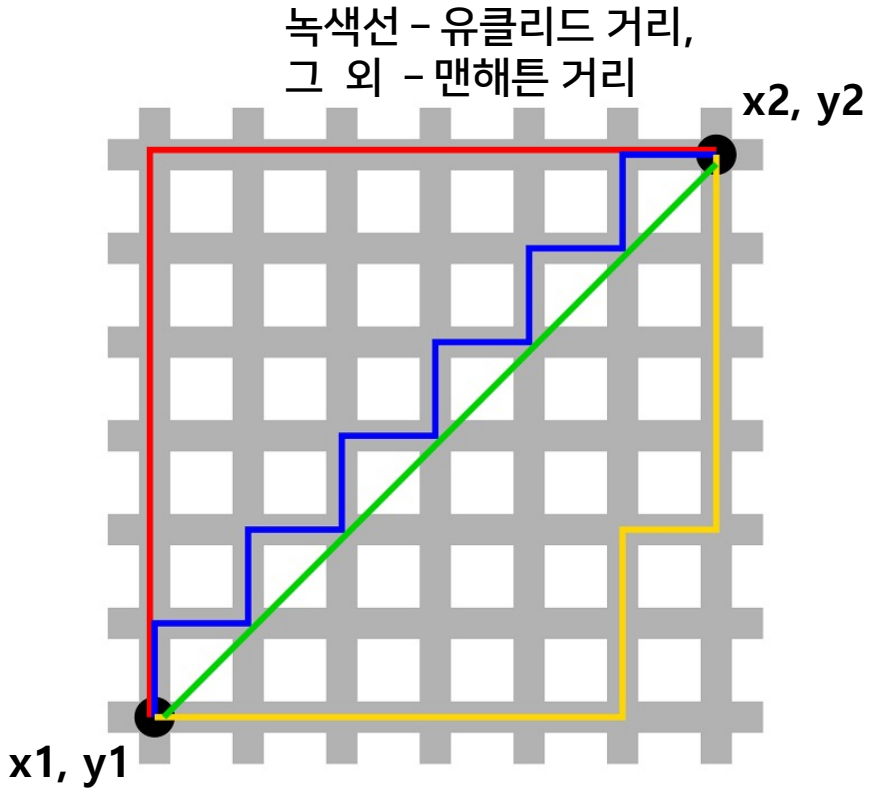
  맨하탄 유사도(Manhattan Similarity)는 맨하탄 거리를 통해 유사도를 측정하는 방법

Manhattan distance = |x1 – x2| + |y1 – y2|

$$\sum_{i=1}^{k} |x_i - y_i|$$

```
From math import*
Def Manhattan_distance(x,y):
      return sum(abs(a-b) for a,b in zip(x,y))
 print manhattan_distance([10,20,10],[10,20,20])

10
[Finished in 0.0s]
```



자카드 유사도(Jaccard similarity)

- **기본 아이디어**

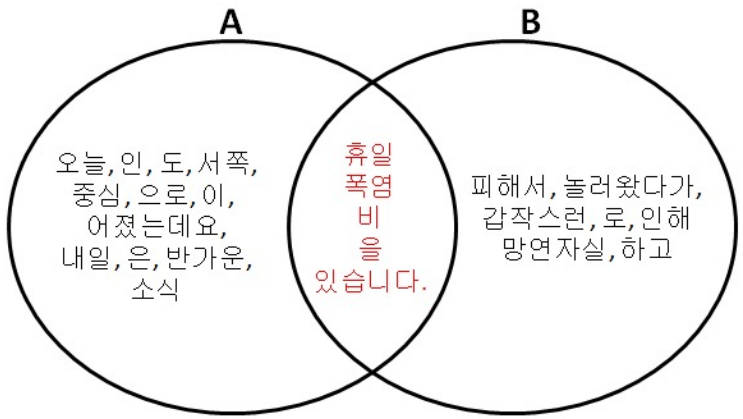
  "합집합에서 교집합의 비율을 구한다면 두 집합 A와 B의 유사도를 구할 수 있다"

  0과 1사이의 값을 가지며, 만약 두 집합이 동일하다면 1, 두 집합의 공통 원소가 없다면 0

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

$$J(doc_1, doc_2) = \frac{doc_1 \cap doc_2}{doc_1 \cup doc_2}$$

두 문서 doc1, doc2 사이의 자카드 유사도 J(doc1,doc2)는 두 집합의 교집합 크기를 두 집합의 합집합 크기로 나눈 값



출처 https://m.blog.naver.com/jaeyoon_95/222206541615

자카드 유사도(Jaccard similarity)

- **예제**

  doc1, doc2 각각 여러 단어를 가진 문서로 정의

```
# 다음과 같은 두 개의 문서가 있습니다.
# 두 문서 모두에서 등장한 단어는 apple과 banana 2개.
doc1 = "apple banana everyone like likey watch card holder"
doc2 = "apple banana coupon passport love you"

# 토큰화를 수행합니다.
tokenized_doc1 = doc1.split()
tokenized_doc2 = doc2.split()

# 토큰화 결과 출력
print(tokenized_doc1)
print(tokenized_doc2)
```

*** 파이썬split 함수 - 괄호 안에 아무것도 넣지 않으면 공백(띄어쓰기, 탭 등)을 기준으로 문자열을 나눔**

```
['apple', 'banana', 'everyone', 'like', 'likey', 'watch', 'card', 'holder']
['apple', 'banana', 'coupon', 'passport', 'love', 'you']
```

자카드 유사도(Jaccard similarity)

예제

- doc1, doc2 의 합집합 – 총 12개

```
union = set(tokenized_doc1).union(set(tokenized_doc2))
print(union)
```

* 파이썬 set.union() – 합집합을 구하는 함수

```
{'card', 'holder', 'passport', 'banana', 'apple', 'love', 'you', 'likey', 'coupon', 'like', 'watch', 'everyone'}
```

- doc1, doc2 의 교집합 – 총 2개

```
intersection = set(tokenized_doc1).intersection(set(tokenized_doc2))
print(intersection)
```

* 파이썬 set.intersection() – 교집합을 구하는 함수

```
{'banana', 'apple'}
```

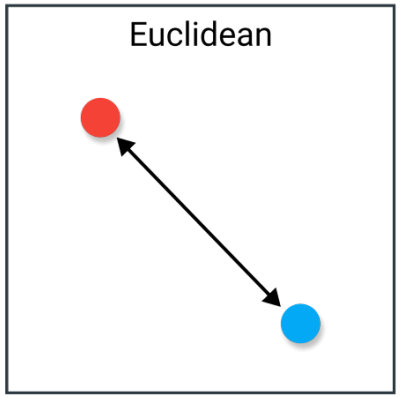
- 자카드 유사도 = 교집합의 수 / 합집합의 수

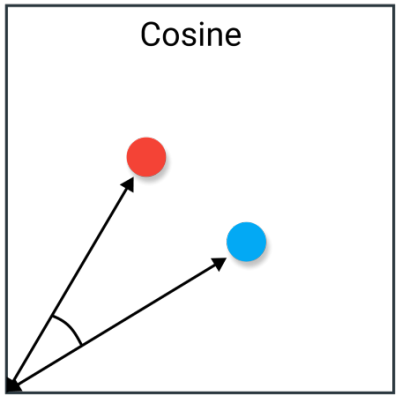
```
print(len(intersection)/len(union)) # 2를 12로 나눔.
```

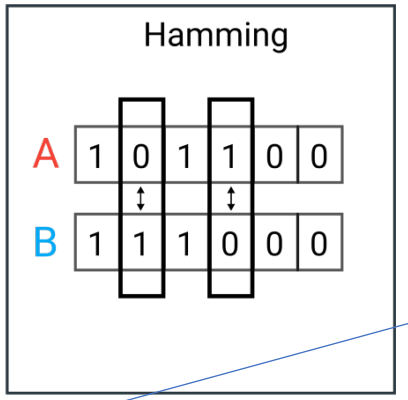
```
0.16666666666666666
```

다양한 유사도 측정 방식

$$D(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2)}$$

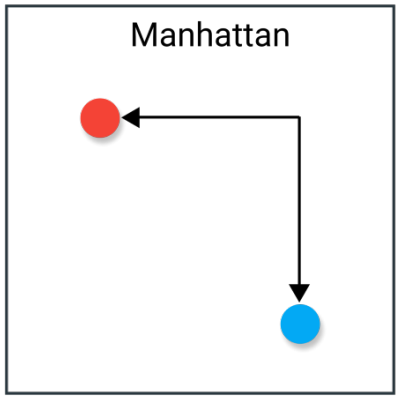


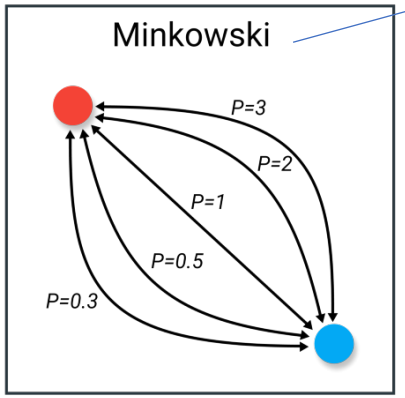


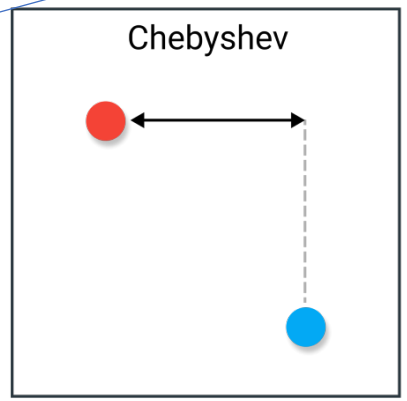


$$D(x,y) = \left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{\frac{1}{p}}$$

$$D(x,y) = \sum_{i=1}^{k}|x_i - y_i|$$

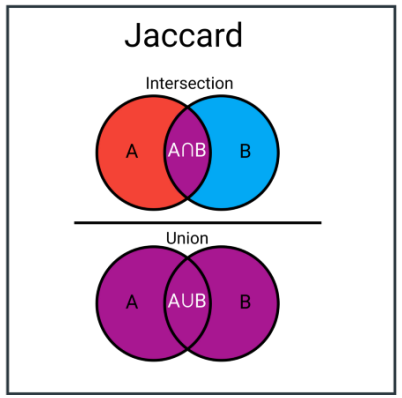


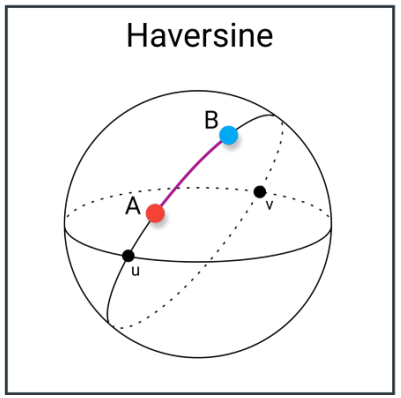


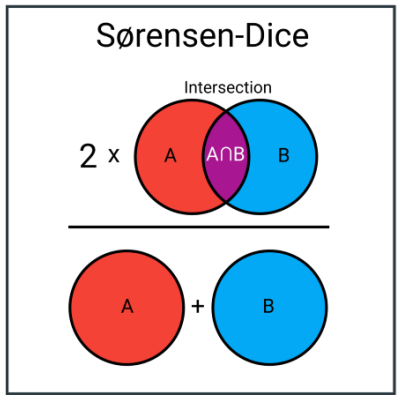


$$D(x,y) = \max_i\left(|x_i - y_i|\right)$$

$$D(x,y) = 1 - \frac{|x \cap y|}{|y \cup x|}$$





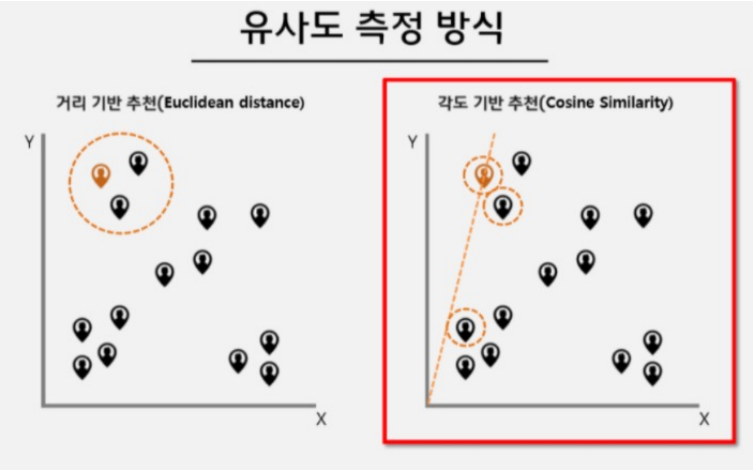


$$D(x,y) = \frac{2\,|x \cap y|}{|x| + |y|}$$

출처 https://towardsdatascience.com/9-distance-measures-in-data-science-918109d069fa

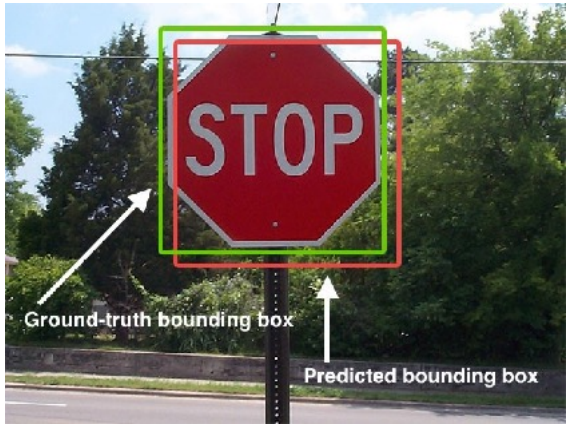
유사도 측정 방법의 비교 및 활용

- **거리를 측정하는 이유 – 데이터간 거리가 바로 유사도를 표현**



- 유클리드/맨해튼/민코프스키 등은 거리의 먼 정도를 측정
- 코사인방식은 방향만 고려, 추천시스템에서 평가 척도의 차이는 고려하지 않음
- Hamming 거리는 주로 맞춤법 검사에 활용

- Jaccard방식은 이미지 분류에도 사용하지만 문서간 단어가 겹치는 것을 측정하여 텍스트 유사 분석에도 활용
- Chebyshev 방식은 한 점에서 다른 점으로 이동시 최소 이동거리의 개념이라 주로 창고 물류분야에서 활용
- Haversine은 구 의 두 점의 거리를 계산하기 때문에 네비게이션에 활용



출처 https://anweh.tistory.com/54https://blog.naver.com/i_am_sangyun/222234083586

4. 레벤슈타인 거리를 이용한 오타 정정

1) 레벤슈타인 거리(Levenshtein Distance)

2) 실습

레벤슈타인 거리를 이용한 오타 정정

레벤슈타인 거리(Levenshtein Distance)

- 두 문자열이 존재할 때, 한 문자열이 다른 문자열로 변경하는 데 필요한 최소 연산 수
- 연산 종류에는 삽입, 삭제, 변경이 있음

Ex) <빅데이터>와 <데이트립> 사이의 레벤슈타인 거리는?

	문자열	연산과정	비용
①	~~빅~~데이터	삭제	1
②	데이~~터~~ 트	변경	1
③	데이트립	삽입	1

→ 삽입, 삭제, 변경 연산이 각 1번씩 사용되므로 3

레벤슈타인 거리를 이용한 오타 정정

레벤슈타인 거리 계산 방법

① 문자열을 2차원 배열로 나타낸다

- 문자열 변경에 필요한 레벤슈타인 거리를 점진적으로 구해 나아간다

문자열의 첫 칸은 공백으로 비워둔다

	ø	빅	데	이	터
ø					
데					
이					
트					
립					

레벤슈타인 거리를 이용한 오타 정정

■ 레벤슈타인 거리 계산 방법

② (0,0)의 비용은 0으로 초기화해주고, 이를 기준으로 첫 행과 첫 열의 값은 1씩 늘려나간다

- 한 글자씩 삽입하는 연산이 일어남을 의미함

빈 문자열(ø)을 '빅데이터'라는 문자열로 변환하기 위해선 한 글자씩 삽입하는 연산이 시행됨을 의미

	ø	빅	데	이	터
ø	0	1	2	3	4
데	1				
이	2				
트	3				
립	4				

레벤슈타인 거리를 이용한 오타 정정

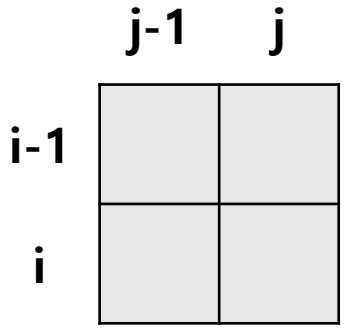
- **레벤슈타인 거리 계산 방법**

③ 다음 규칙을 이용해 2차원 배열 값을 모두 채울 때까지 반복한다

$$\text{Cost} = \begin{cases} 0 & \textit{if (i-1)번째 문자열 = (j-1)번째 문자열,} \\ 1 & \textit{otherwise.} \end{cases}$$

- **(i, j)** 값은 다음 세 값 중 최소값에 해당함

1) **(i-1, j) + 1** **(삽입 비용)**
2) **(i, j-1) + 1** **(삭제 비용)**
3) **(i-1,j-1) + Cost** **(수정 비용)**



	ø	빅	데	이	터
ø	0	1	2	3	4
데	1	1	1	2	3
이	2	2	2	1	2
트	3	3	3	2	2
립	4	4	4	3	3

레벤슈타인 거리를 이용한 오타 정정

- **레벤슈타인 거리 계산 방법**

③ 다음 규칙을 이용해 2차원 배열 값을 모두 채울 때까지 반복한다

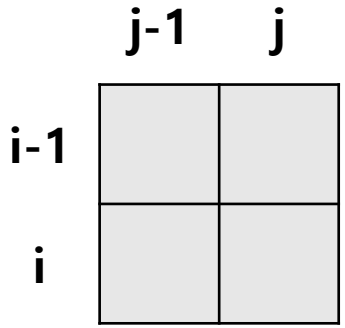
$$Cost = \begin{cases} 0 & \textit{if (i-1)번째 문자열 = (j-1)번째 문자열,} \\ 1 & \textit{otherwise.} \end{cases}$$

- **(i, j)** 값은 다음 세 값 중 최소값에 해당함

1) **(i-1, j) + 1** **(삽입 비용)**

2) **(i, j-1) + 1** **(삭제 비용)**

3) **(i-1,j-1) + Cost** **(수정 비용)**



	ø	빅	데	이	터
ø	0	1	2	3	4
데	1	1	1	2	3
이	2	2	2	1	2
[illegible]	[illegible]	3	3	2	2
립	4	4	4	3	3

Cost = 1 (이 != 데)

1) 1 + 1 = 2
2) 3 + 1 = 4
3) 2 + 1 = 3

→ 최소값 2

레벤슈타인 거리를 이용한 오타 정정

- **레벤슈타인 거리 계산 방법**

③ 다음 규칙을 이용해 2차원 배열 값을 모두 채울 때까지 반복한다

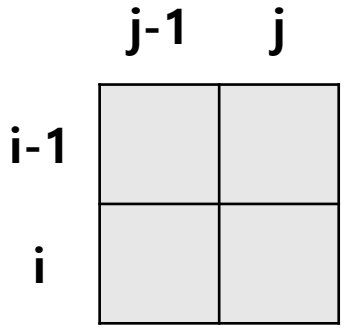
$$Cost = \begin{cases} 0 & \textit{if (i-1)번째 문자열 = (j-1)번째 문자열,} \\ 1 & \textit{otherwise.} \end{cases}$$

- **(i, j)** 값은 다음 세 값 중 최소값에 해당함

1) **(i-1, j) + 1** **(삽입 비용)**

2) **(i, j-1) + 1** **(삭제 비용)**

3) **(i-1,j-1) + Cost** **(수정 비용)**



	ø	빅	데	이	터
ø	0	1	2	3	4
데	1	1	1	2	3
이	2	2	2	1	2
트	3	3	3	2	2
립	4	4	4	3	3

Cost = 1 (데 != 이)

1) 3 + 1 = 4
2) 1 + 1 = 2
3) 2 + 1 = 3

→ 최소값 2

레벤슈타인 거리를 이용한 오타 정정

- **레벤슈타인 거리 계산 방법**

③ 다음 규칙을 이용해 2차원 배열 값을 모두 채울 때까지 반복한다

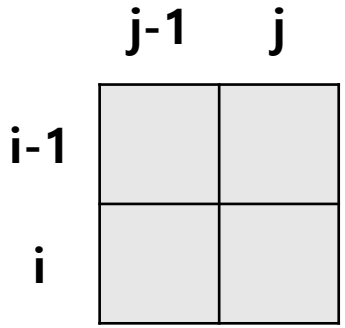
$$Cost = \begin{cases} 0 & \textit{if (i-1)번째 문자열 = (j-1)번째 문자열,} \\ 1 & \textit{otherwise.} \end{cases}$$

- **(i, j)** 값은 다음 세 값 중 최소값에 해당함

1) **(i-1, j) + 1** **(삽입 비용)**

2) **(i, j-1) + 1** **(삭제 비용)**

3) **(i-1,j-1) + Cost** **(수정 비용)**



	ø	빅	데	이	터
ø	0	1	2	3	4
데	1	1	1	2	3
이	2	2	2	1	2
트	3	3	3	2	2
립	4	4	4	3	3

Cost = 0 (데 = 데)

1) 2 + 1 = 3
2) 2 + 1 = 3
3) 1 + 0 = 1

→ 최소값 1

레벤슈타인 거리를 이용한 오타 정정

- **레벤슈타인 거리 계산 방법**

  ④ 우측하단 값이 두 문자열 사이의 레벤슈타인 거리가 된다

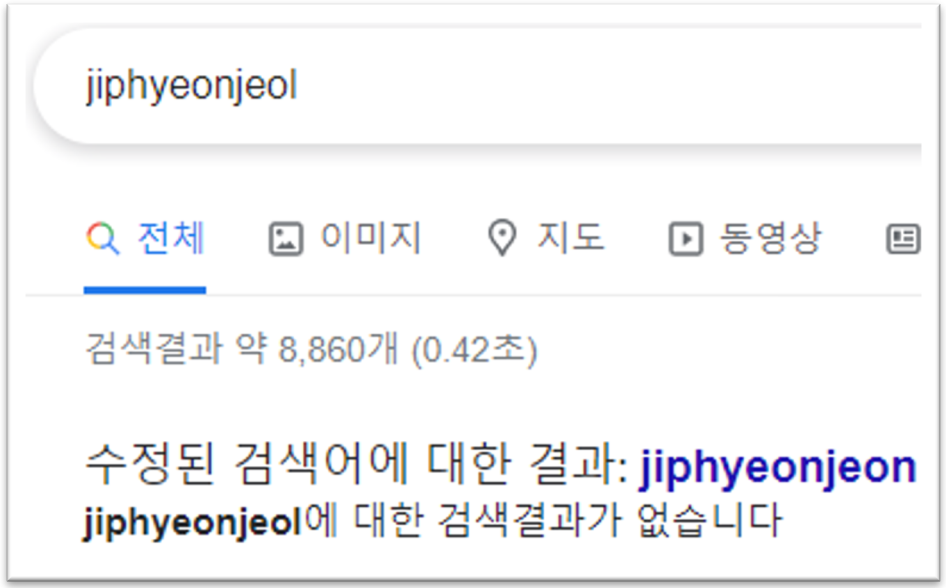
	ø	빅	데	이	터
ø	0	1	2	3	4
데	1	1	1	2	3
이	2	2	2	1	2
트	3	3	3	2	2
립	4	4	4	3	3

레벤슈타인 거리를 이용한 오타 정정

■ 레벤슈타인 거리를 이용한 오타정정

- 검색엔진 DB에 철자가 올바른 단어가 정의된 사전이 존재한다고 가정할 때
- 만일 이용자가 검색한 단어가 사전에 존재하지 않는 경우?

▶ 검색어와 레벤슈타인 거리가 가장 작은 단어를 반환해줌



- 올바른 단어가 정의된 사전에 jiphyeonjeol이라는 단어가 존재하지 않음
- Jiphyeonjeol이라는 단어와 레벤슈타인 거리가 가장 작은 단어인 jiphyeonjeon이라는 단어를 반환해줌

레벤슈타인 거리를 이용한 오타 정정 (실습)

레벤슈타인 거리 코드

```
def leven(aText,bText):
  '''
  입력값: 문자열 2개
  출력값: 두 문자열 사이의 레벤슈타인 거리'''
  # (문자열 길이 + 1 ) * (문자열 길이 + 1)의 2D 배열을 만들어 준다
  aLen = len(aText)+1
  bLen = len(bText)+1
  array = [ [] for a in range(aLen) ]
  for i in range(aLen):
    array[i] = [0 for a in range(bLen)]
  # (첫 행과 첫 열을 초기화해준다
  for i in range(bLen):
    array[0][i] = i
  for i in range(aLen):
    array[i][0] = i
  cost = 0
  for i in range(1,aLen):
    for j in range(1,bLen):
      if aText[i-1] != bText[j-1]: # (i-1)번째 문자와 (j-1)번째 문자가 다를 시 Cost = 1
        cost = 1
      else : cost = 0
      addNum = array[i-1][j] + 1 # 1번 케이스
      minusNum = array[i][j-1] + 1 # 2번 케이스
      modiNum = array[i-1][j-1]+cost # 3번 케이스
      minNum = min([addNum,minusNum,modiNum]) # 1, 2, 3 케이스 중 최소값을 구해준다
      array[i][j] = minNum # 최소값을 (i, j)에 대입해준다
  result = array[aLen-1][bLen-1] # 최종 레벤슈타인 거리를 반환해줌
  return result
```

출처 https://hoony-gunputer.tistory.com/entry/레벤슈타인-거리를-이용해서-두-문장-비교하기

레벤슈타인 거리를 이용한 오타 정정 (실습)

- **색채어 오타 정정하기**
  - 검색엔진에 색채어를 검색했을 때 오타가 난 경우, 이를 정정해주는 코드를 작성한다고 가정

레벤슈타인 거리를 이용한 오타 정정 (실습)

색채어 사전

```
dictionary = ['Mahogany', 'Fuzzy Wuzzy Brown', 'Chestnut', 'Red Orange', 'Sunset Orange', 'Bittersweet', 'Melon', 'Outrageous Orange', 'Vivid Tangerine',
              'Burnt Sienna', 'Brown', 'Sepia', 'Orange', 'Burnt Orange', 'Copper', 'Mango Tango', 'Atomic Tangerine', 'Beaver', 'Antique Brass', 'Desert Sand',
              'Raw Sienna', 'Tumbleweed', 'Tan', 'Peach', 'Macaroni and Cheese', 'Apricot', 'Neon Carrot', 'Almond', 'Yellow Orange', 'Gold', 'Shadow',
              'Banana Mania', 'Sunglow', 'Goldenrod', 'Dandelion', 'Yellow', 'Green Yellow', 'Spring Green', 'Olive Green', 'Laser Lemon', 'Unmellow Yellow',
              'Canary', 'Yellow Green', 'Inch Worm', 'Asparagus', 'Granny Smith Apple', 'Electric Lime', 'Screamin Green', 'Fern', 'Forest Green', 'Sea Green',
              'Green', 'Mountain Meadow', 'Shamrock', 'Jungle Green', 'Caribbean Green', 'Tropical Rain Forest', 'Pine Green', 'Robin Egg Blue', 'Aquamarine',
              'Turquoise Blue', 'Sky Blue', 'Outer Space', 'Blue Green', 'Pacific Blue', 'Cerulean', 'Cornflower', 'Midnight Blue', 'Navy Blue', 'Denim', 'Blue',
              'Periwinkle', 'Cadet Blue', 'Indigo', 'Wild Blue Yonder', 'Manatee', 'Blue Bell', 'Blue Violet', 'Purple Heart', 'Royal Purple','Purple Mountains',
              'Majesty', 'Violet (Purple)', 'Wisteria', 'Vivid Violet', 'Fuchsia', 'Shocking Pink', 'Pink Flamingo', 'Plum', 'Hot Magenta', 'Purple Pizzazz',
              'Razzle Dazzle Rose', 'Orchid', 'Red Violet', 'Eggplant', 'Cerise', 'Wild Strawberry', 'Magenta', 'Lavender', 'Cotton Candy', 'Violet Red',
              'Carnation Pink', 'Razzmatazz', 'Piggy Pink', 'Jazzberry Jam', 'Blush', 'Tickle Me Pink', 'Pink Sherbet', 'Maroon', 'Red', 'Radical Red', 'Mauvelous',
              'Wild Watermelon','Scarlet',  'Salmon', 'Brick Red', 'White', 'Timberwolf', 'Silver', 'Gray', 'Black']
```

색채어 오타 정정 함수 작성

```
def Color_Name_Corrector(color, dictionary):
  '''입력값
  - color: 검색하고자 하는 색채어
  - dictionary: 색채어가 정의된 사전'''
  distances = []
  color = color.lower()

  for word in dictionary: # 사전 속 모든 단어와의 레벤슈타인 거리를 구한다
    word = word.lower()
    distance = leven(color, word)

    if distance == 0: # 검색어가 사전에 있는 경우 곧바로 검색어를 반환함
      return [color]

    else:
      distances.append(distance)

  min_dist = min(distances) # 레벤슈타인 거리 최소값을 구함
  answer = [dictionary[idx] for idx, x in enumerate(distances) if x == min_dist] # 레벤슈타인 거리가 최소인 단어를 구함(1개 이상 반환 가능)

  return answer
```

레벤슈타인 거리를 이용한 오타 정정 (실습)

색채어 오타 정정 함수 결과

```
1 Color_Name_Corrector('grayck', dictionary)
```

```
['gray']
```

→ ‘grayck’라는 단어와 레벤슈타인 거리가 가장 짧은 ‘gray’로 반환

```
1 Color_Name_Corrector('red', dictionary)
```

```
['red']
```

→ 사전에 존재하는 단어

```
1 Color_Name_Corrector('gredy', dictionary)
```

```
['green', 'red', 'gray']
```

→ ‘gredy’라는 단어와 레벤슈타인 거리가 짧은 단어가 3가지 존재함

감사합니다