8. 텍스트 분석

goblurry · 어제

데이터분석 머신러닝 파완머

📊 파이썬 머신러닝 완벽 가이드 (위키북스, 개정 2판) 교재의 8장을 바탕으로 학습한 내용을 정리한 포스트입니다.

목차

들어가며

텍스트 분석이란 텍스트 마이닝이라고도 불리며, **비정형인 텍스트에서 유의미한 정보를 추출하기 위해 사용되는 기술**이다. 텍스트 분석이 주로 수행하는 작업은 비지니스 인텔리전스나 예측 분석과 같은 분석 작업인데, 이는 머신러닝, 언어이해, 통계 등을 활용해 모델을 수립하고 정보를 추출함으로써 이루어진다.

머신러닝 기술의 발전에 따라 텍스트 분석도 발전하고 있다.

텍스트 분석이 사용되는 영역

1. 텍스트 분류(Text classification or categorization): 문서가 특정 분류 또는 카테고리에 속하는 것을 예측하는 기법. 지도학습 적용. ex) 특정 신문 기사 내용이 정치/연예/사회 중 어떤 영역에 속하는지?
2. 감성 분석: 텍스트에서 나타나는 감정 등 주관적 요소를 분석하는 기법. 소셜 미디어 감정 분석, 영화 및 제품에 대한 리뷰, 여론조사 의견 등에서 활용되며, 텍스트 분석에서 가장 활발한 분야. 지도학습, 비지도학습 모두 이용 가능하다.
3. 텍스트 요약: 텍스트 내 주제나 중심 사상 추출하는 기법. 토픽 모델링이 대표적.
4. 텍스트 군집화와 유사도 측정: 비슷한 유형의 문서에 대해 군집화를 수행하고, 문서들 간의 유사도를 측정해 비슷한 문서끼리 모으는 기법. 비지도학습으로 텍스트 분류를 수행하는 방법의 일종이다.

텍스트 분석 이해

지금까지의 머신러닝 모델은 주어진 **정형 데이터**를 기반으로 모델을 수립하고 예측을 수행했고, 알고리즘은 숫자형의 피처 기반 데이터를 입력받았다. 하지만 텍스트는 **비정형 데이터**이다. 텍스트를 머신러닝에 적용하기 위해서는, 이러한 텍스트 데이터를 어떻게 피처 형태로 추출하고, 추출된 피처에 유의미한 값을 부여할지가 아주 중요한 요소이다.

텍스트를 Word 기반의 다수의 피처로 추출하고, 이 피처에 빈도수 같은 숫자값을 부여할 경우 텍스트는 단어의 조합인 벡터값으로 표현될 수 있다. ⟹ **피처 벡터화 또는 피처 추출**
+) 피처 벡터화 방법: Bag of Words(BOW), Word2Vec

한마디로, 머신러닝 모델로 텍스트 분석을 수행할 때 핵심은 텍스트를 '벡터값을 갖는 피처로 변환하는 것'이다.

텍스트 분석 수행 프로세스

머신러닝 기반의 텍스트 분석 프로세스는 다음과 같다.

1. 텍스트 사전 준비 작업(텍스트 전처리)
   텍스트 > 피처 변환 전 미리 대소문자 변경, 특수문자 삭제 등의 클렌징 작업, 단어 등 토큰화 작업, 무의미한 단어(Stop word) 제거, 어근 추출(Stemming/Lemmatization) 등 텍스트 정규화 작업 수행.

2. 피처 벡터화/추출
   1에서 가공된 텍스트로부터 피처를 추출하고, 여기에 벡터값을 할당함.
   - BOW(Count 기반 벡터화, TF-IDF 기반 벡터화)
   - Word2Vec

3. ML 모델 수립 및 학습/예측/평가
   피처 벡터화된 데이터 세트에 ML 모델을 적용해 학습, 예측 및 평가,를 수행한다.

파이썬 기반 NLP, 텍스트 분석 패키지
파이썬 기반에서 위의 프로세스를 지원하는 라이브러리: NLTK, gensim, SpaCy
NLTK: 파이썬 대표 NLP 패키지. 하지만 수행 속도 측면에서 아쉬운 부분이 있어서 실제 대량 데이터 기반에서는 활용도가 떨어짐.
Gensim: 토픽 모델링(텍스트 요약의 일종)에서 가장 두각을 나타내는 패키지.
Spacy: 수행 성능이 뛰어나서 가장 주목 받고 있는 NLP 패키지. 최근 사용 사례 증가.

텍스트 전처리: 텍스트 정규화

텍스트를 피처로 변환하기 전 사전 가공 작업이 필요하고, 매우 중요하다.

텍스트 정규화: ML 알고리즘·NLP 애플리케이션에 텍스트를 입력 데이터로 사용하기 위해 클렌징, 정제, 토큰화, 어근화 등의 사전 작업을 수행하는 것

1. 클렌징: 불필요한 문자, 기호를 제거하는 작업. (HTML, XML 태그 및 특정 기호 등)

2. 토큰화: 문장 토큰화(문서에서 문장 분리) / 단어 토큰화(문장에서 단어 분리)

- 문장 토큰화: 문장 마침표나 개행문자(\n) 등 문장 마지막을 뜻하는 기호에 따라 문장을 분리. NLTK의 `sent_tokenize()` 이용해 토큰화 가능. 토큰화 후 반환되는 것은 list 객체이다.
- 단어 토큰화: 문장을 단어로 토큰화하는 과정. 기본적으로 공백, 콤마, 마침표, 개행문자 등으로 단어를 분리한다. 정규 표현식을 이용해 다양한 유형으로 수행할 수도 있다. NLTK의 `word_tokenize()` 를 이용한다. 마찬가지로 list 객체로 반환된다.
- 문장을 단어별로 토큰화하는 경우 문맥적 의미는 당연히 무시된다. 이를 조금이라도 해결하고자 도입된 게 n-gram이다. 연속된 n개의 단어를 하나의 토큰화 단위로 분리해내는 것. ("Agent Smith knocks the door" 라는 문장을 2-gram으로 만든다면 (Agent, Smith), (Smith, knocks), (knocks, the), (the, door)과 같이 연속 2개 단어를 순차적으로 이동하며 단어를 토큰화하는

것이다.

3. 필터링/스톱 워드 제거/철자 수정: is, the, a... 처럼 문장의 필수 요소지만 문맥적으로 의미가 없는 단어들을 스톱 워드라고 한다. 빈번하게 등장하기 때문에 이것들을 사전 제거하지 않으면 이것들이 중요한 단어로 인식될 수 있다. NLTK는 다양한 언어의 스톱 워드를 제공한다. (nltk.download('stopwords')로 그 목록을 확인할 수 있다.)

4. Stemming, Lemmatization: 문법적·의미적으로 변화하는 단어의 원형을 찾는 작업. 많은 언어에서 문법 요소에 따라 단어의 형태가 바뀌기 때문에 필요한 과정이다. (예: work > works, wordking, worked...)
Stemming과 Lemmatizaiton은 목적이 같지만 후자가 더 정교하고 의미론적 기반에서 단어의 원형을 찾는다.

- Stemming: 일반적 방법/단순화된 방법을 적용해 원래 단어에서 일부 철자가 훼손된 어근 단어를 추출 (NLTK 제공 Stemmer: Porter, Lancaster, Snowball Stemmer)
- Lemmatization: 품사 같은 문법적 요소와 더 의미적인 부분을 감안해 정확한 철자로 된 어근 단어 추출 (변환에 더 오랜 시간 소요, NLTK 제공 Lemmatization: WordNetLemmatizer)

Stemming: LancasterStemmer 사용
진행형, 3인칭 단수, 과거형에 따른 동사, 비교/최상에 따른 형용사 변화
NLTK: LancasterStemmer()과 같이 필요한 스테머 객체를 생성하고 이 객체의 stem('단어') 메서드를 호출하면 '단어'의 스테밍이 가능함.

```
from nltk.stem import LancasterStemmer
stemmer = LancasterStemmer()

print(stemmer.stem('working'), stemmer.stem('works'), stemmer.stem('worked'))
print(stemmer.stem('amusing'), stemmer.stem('amuses'), stemmer.stem('amused'))
print(stemmer.stem('happier'), stemmer.stem('happiest'))
print(stemmer.stem('fancier'), stemmer.stem('fanciest'))
print(stemmer.stem('better'), stemmer.stem('good'), stemmer.stem('best')) # 궁금해서 추가
```

✓ 0.0s

```
work work work
amus amus amus
happy happiest
fant fanciest
bet good best
```

Stemming을 수행한 결과이다. work의 경우 'work'에 s, ed, ing이 붙는 단순 변화이므로 원형을 잘 인식하지만, amuse는 각 변화가 amus+ 형태로 이루어져서 원형 단어로 amus를 인식한다. 형용사도 단어의 정확한 원형을 찾지 못하고 철자가 다른 어근 단어로 인식한다.

Lemmatization: WordNetLemmatizer 이용.
일반적으로 더 정확하게 원형 단어를 추출하기 위해 단어의 **품사**를 입력해야 한다. `lemmatize`의 파라미터로 동사는 v, 형용사는 a를 입력한다.

```
from nltk.stem import WordNetLemmatizer
import nltk
nltk.download('wordnet')

lemma = WordNetLemmatizer()
print(lemma.lemmatize('amusing', 'v'), lemma.lemmatize('amuses', 'v'), lemma.lemmatize('amused', 'v'))
print(lemma.lemmatize('happy', 'a'), lemma.lemmatize('happier', 'a'), lemma.lemmatize('happiest', 'a'))
print(lemma.lemmatize('fancier', 'a'), lemma.lemmatize('fanciest', 'a'))
print(lemma.lemmatize('better', 'a'), lemma.lemmatize('good', 'a'), lemma.lemmatize('best', 'a'))
```

(function) lemmatize: Any

✓ 2.2s

```
[nltk_data] Downloading package wordnet to /Users/hyun/nltk_data...
amuse amuse amuse
happy happy
fancy fancy
good good best
```

Lemmatization 수행 결과, Stemming보다 추출된 어근 단어의 정확도가 높은 것을 확인할 수 있다.

Bag of Words: BOW

BOW 모델: 피처를 벡터화할 때, 문서가 가지는 모든 단어를 문맥/순서를 무시하고 일괄적으로 **빈도 값**을 부여해 피처 값을 추출하는 모델 (단어 발생 횟수에 기반)

[BOW의 단어 수 기반 피처 추출]
문장 1: My wife likes to watch baseball games and my daughter likes to watch baseball games too.
문장 2: My wife likes to play baseball.

이라는 두 문장이 있을 때, 모든 단어에서 중복을 제거하고 각 단어를 칼럼 형태로 나타낸 뒤, 각 단어에 고유 인덱스를 부여한다. (and=0, baseball=1, ..., wife=10)
개별 문장에서 해당 단어의 발생 횟수를 아래 사진처럼 각 단어에 기재한다.

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

→ 문장 1에서 baseball은 2회 나타남

장점: 쉽고 빠른 구축. 예상보다 문서의 특징을 잘 나타내 여러 분야에서 높은 활용.
단점: (1) 문맥 의미 반영 부족 (2) 희소 행렬 문제

희소 행렬: 단어를 전부 추출하면 칼럼 규모가 커지게 되는데, 이 단어들이 여러 문서에서 중복되어 나타나는 경우보다 그렇지 않은 경우가 더 많다. 즉, 대부분의 데이터가 0 값으로 채워지는, 희소 행렬 형태의 데이터 세트가 형성된다. > ML 알고리즘의 수행 시간과 예측 성능을 떨어뜨림.

BOW 피처 벡터화

위에서 말했듯 BOW 모델에서 피처 벡터화를 수행한다는 것은 모든 문서에서 모든 단어를 칼럼 형태로 나열하고, 각 문서에서 해당 단어의 횟수나 정규화된 빈도를 값으로 부여하는 데이터 세트 모델로 변경하는 것이다.
M개의 문서가 있고, 여기서 추출되어 나열된 단어 수가 N개라면, 이 행렬은 MxN 개의 단어 피처로 구성된다.

BOW의 피처 벡터화 방식: (1) 카운트 기반의 벡터화 (2) TF-IDF (Term Frequency - Inverse Document Frequency) 기반의 벡터화

카운트 기반: 단어 피처에 값을 부여할 때 말 그대로 발생 횟수를 부여하는 경우. 카운트 값과 단어의 중요도가 비례한다. 하지만 이 경우 문제가 문서의 특징을 반영하기보다 언어 특성상 자주 사용될 수밖에 없는 단어에까지 높은 값을 부여하게 된다는 것이다.

TF-IDF: 카운트 기반 벡터화의 문제점을 보완하기 위한 벡터화 방식. **개별** 문서에서 자주 나타나는 단어에는 높은 가중치를 주고, **모든** 문서에서 자주 나타나는 문서에는 패널티를 준다. 후자의 경우 문서를 특정짓기보다는 언어 특성상 범용적으로 자주 쓰이는 단어일 가능성이 높기 때문이다. 이렇게 가중치의 균형을 맞추는 게 TF-IDF 기반의 벡터화이고, 문서 텍스트가 길고 개수가 많을 때 카운트 방식보다 예측 성능을 높일 수 있다.

사이킷런의 카운트 및 TF-IDF 벡터화 구현: `CountVectorizer` , `TfidfVectorizer`

파라미터 명	파라미터 설명
max_df	전체 문서에 걸쳐서 너무 높은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터입니다. 너무 높은 빈도수를 가지는 단어는 스톱 워드와 비슷한 문법적인 특성으로 반복적인 단어일 가능성이 높기에 이를 제거하기 위해 사용됩니다. max_df = 100과 같이 정수 값을 가지면 전체 문서에 걸쳐 100개 이하로 나타나는 단어만 피처로 추출합니다. Max_df = 0.95와 같이 부동소수점 값(0.0 ~ 1.0)을 가지면 전체 문서에 걸쳐 빈도수 0~95%까지의 단어만 피처로 추출하고 나머지 상위 5%는 피처로 추출하지 않습니다.
min_df	전체 문서에 걸쳐서 너무 낮은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터입니다. 수백~수천 개의 전체 문서에서 특정 단어가 min_df에 설정된 값보다 적은 빈도수를 가진다면 이 단어는 크게 중요하지 않거나 가비지(garbage)성 단어일 확률이 높습니다. min_df = 2와 같이 정수 값을 가지면 전체 문서에 걸쳐서 2번 이하로 나타나는 단어는 피처로 추출하지 않습니다. min_df = 0.02와 같이 부동소수점 값(0.0 ~ 1.0)을 가지면 전체 문서에 걸쳐서 하위 2% 이하의 빈도수를 가지는 단어는 피처로 추출하지 않습니다.
max_features	추출하는 피처의 개수를 제한하며 정수로 값을 지정합니다. 가령 max_features = 2000으로 지정할 경우 가장 높은 빈도를 가지는 단어 순으로 정렬해 2000개까지만 피처로 추출합니다.
stop_words	'english'로 지정하면 영어의 스톱 워드로 지정된 단어는 추출에서 제외합니다.
n_gram_range	Bag of Words 모델의 단어 순서를 어느 정도 보강하기 위한 n_gram 범위를 설정합니다. 튜플 형태로 (범위 최솟값, 범위 최댓값)을 지정합니다. 예를 들어 (1, 1)로 지정하면 토큰화된 단어를 1개씩 피처로 추출합니다. (1, 2)로 지정하면 토큰화된 단어를 1개씩(minimum 1), 그리고 순서대로 2개씩(maximum 2) 묶어서 피처로 추출합니다.
analyzer	피처 추출을 수행한 단위를 지정합니다. 당연히 디폴트는 'word'입니다. Word가 아니라 character의 특정 범위를 피처로 만드는 특정한 경우 등을 적용할 때 사용됩니다.
token_pattern	토큰화를 수행하는 정규 표현식 패턴을 지정합니다. 디폴트 값은 '\b\w\w+\b로, 공백 또는 개행 문자 등으로 구분된 단어 분리자(\b) 사이의 2문자(문자 또는 숫자, 즉 영숫자) 이상의 단어(word)를 토큰으로 분리합니다. analyzer= 'word'로 설정했을 때만 변경 가능하나 디폴트 값을 변경할 경우는 거의 발생하지 않습니다.
tokenizer	토큰화를 별도의 커스텀 함수로 이용시 적용합니다. 일반적으로 CountTokenizer 클래스에서 어근 변환 시 이를 수행하는 별도의 함수를 tokenizer 파라미터에 적용하면 됩니다.

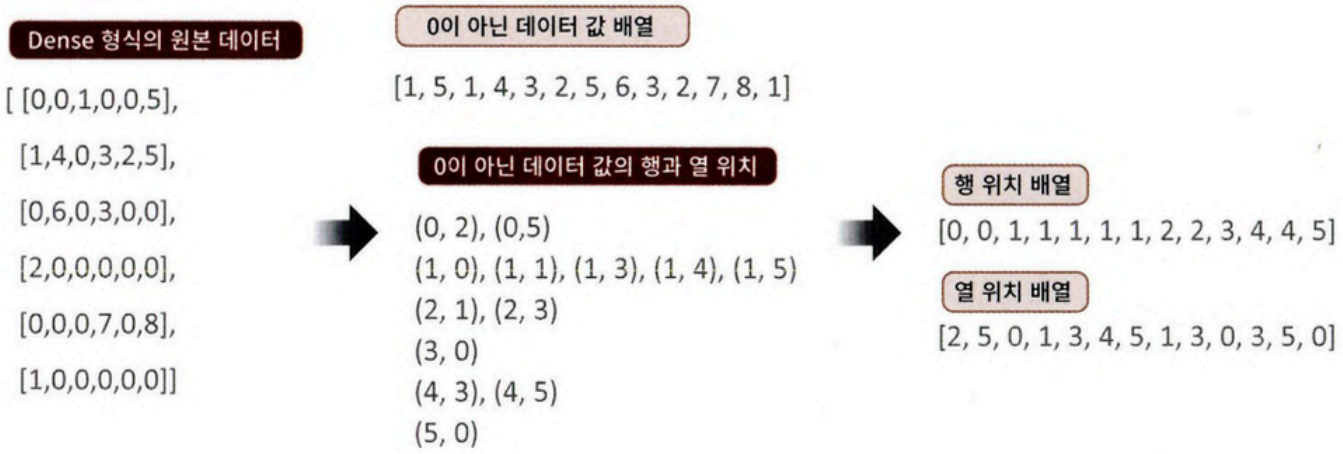
BOW 벡터화를 위한 희소 행렬

위의 두 클래스를 사용해 텍스트를 피처 단위로 벡터화해 변환하고 **CSR** 형태의 희소 행렬을 반환한다. 희소 행렬은 앞서 설명했듯 행렬에 0 값이 너무 많이 할당되어 메모리 공간과 연산 수행 시간을 불필요하게 차지한다는 점에서 문제가 있다. 이러한 희소 행렬을, 물리적으로 적은 공간을 차지하도록 변환해야 한다. 이때 사용되는 방법이 **COO 형식**과 **CSR 형식**이다.

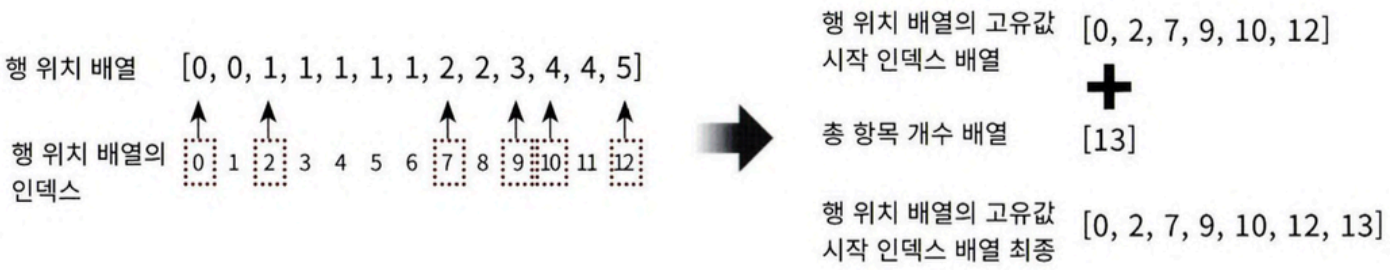
COO 형식 : Coordinate, 즉 좌표 형식. Not 0 데이터를 별도의 데이터 배열에 저장하고, 그 데이터가

가리키는 행과 열의 위치를 또 다른 배열로 저장하는 방식. > 파이썬에서는 Scipy를 이용 - `sparse` 이용해 COO 형식으로 희소 행렬 변환 수행.

CSR 형식: Compressed sparse row 형식. 위의 COO 형식에서는 0이 아닌 데이터의 행과 열 위치를 나타내기 위해 위치 데이터를 사용해야 한다는 문제가 있었다.



사진처럼 순차적인 동일 값이 반복적으로 나타난다는 것이다. CSR 방식은 행 위치 배열 내에 잇는 고유한 값의 **시작 위치**만 별도의 위치 배열로 가지는 변환 방식이다.



이렇게 하면 고유값의 시작 위치만 알면 되기 때문에 COO 방식보다 메모리가 적게 들고 연산 수행 속도도 향상된다. > Scipy의 `csr_matrix` 클래스 이용.

감성 분석

감성 분석은 용어 그대로 문서의 주관적인 감성, 의견, 기분 등을 파악하기 위한 방법으로 주로 온라인

리뷰나 소셜 미디어 등에서 활용된다. 문서 내 텍스트가 나타내는 여러 **주관적 단어와 문맥을 기반으로 감성 수치**를 계산한다.
감성 지수: 긍정 감성 지수 & 부정 감성 지수. 이 지수를 합산해 감성을 결정한다.
감성 분석은 머신러닝의 관점에서 지도학습 & 비지도학습으로 나눌 수 있다.

- 지도학습: 학습 데이터와 타겟 레이블 값을 기반으로 감성 분석 학습을 수행하고, 이를 기반으로 다른 데이터의 감성 분석을 예측하는 방법. (일반 텍스트 기반 분류와 유사하다.)
- 비지도학습: `Lexicon` 이라는 일종의 감성 어휘 사전을 이용하는데, 이는 감성 분석을 위한 용어와 문맥에 대한 여러 정보를 가지고 있어서 이를 활용해 문서의 긍정/부정 감성 여부를 판단한다.

비지도학습 기반 감성 분석: Lexicon

Lexicon: 감성을 분석하기 위해 지원하는 감성 어휘 사전. 긍정 감성 또는 부정 감성의 정도를 의미하는 수치를 가지며, 이를 **감성 지수 Polarity score**라고 한다.
이 감성지수는 단어 위치, 주변 단어, 품사 등을 참고해 결정되며, 대표적으로 NLTK 패키지에 포함되어 있다. > `WordNet`
일반적으로 많은 감성 분석용 데이터는 결정된 레이블 값이 없어서, 렉시콘이 유용하게 사용된다.

SentiWordNet: NLTK 패키지의 WordNet과 유사하게 감성 단어 전용의 WordNet을 구현한 것입니다. WordNet의 Synset 개념을 감성 분석에 적용한 것입니다. WordNet의 Synset별로 3가지 감성 점수(sentiment score)를 할당합니다. 긍정 감성 지수, 부정 감성 지수, 객관성 지수가 그것입니다. 긍정 감성 지수는 해당 단어가 감성적으로 얼마나 긍정적인가를, 부정 지수는 얼마나 감성적으로 부정적인가를 수치로 나타낸 것입니다. 객관성 지수는 긍정/부정 감성 지수와 완전히 반대되는 개념으로 단어가 감성과 관계없이 얼마나 객관적인지를 수치로 나타낸 것입니다. 문장별로 단어들의 긍정 감성 지수와 부정 감성 지수를 합산하여 최종 감성 지수를 계산하고 이에 기반해 감성이 긍정인지 부정인지를 결정합니다.

VADER: 주로 소셜 미디어의 텍스트에 대한 감성 분석을 제공하기 위한 패키지입니다. 뛰어난 감성 분석 결과를 제공하며, 비교적 빠른 수행 시간을 보장해 대용량 텍스트 데이터에 잘 사용되는 패키지입니다.

Pattern: 예측 성능 측면에서 가장 주목받는 패키지입니다. 아쉽게도 현재 기준으로 파이썬 3.X 버전에서 호환이 되지 않고, 파이썬 2.X 버전에서만 동작합니다. 이 책에서는 사용 예제를 소개하지는 않습니다만, 감성 분석에 관심이 많은 사람이라면 적용해 보는 것도 좋습니다.

더 자세한 개념과 실습 내용은
https://github.com/goblurry/9th-ML/blob/6cfc7b246aec665e951ca007113c9939c089a3cd/week12_%EC%98%88%EC%8A%B5%EA%B3%BC%EC%A0%9C_%EB%85%B8%ED%98%84%EC%84%A0.ipynb 에서 확인할 수 있다.

추가 예정...

goblurry

이전 포스트

7. 군집화

0개의 댓글

댓글을 작성하세요

댓글 작성