* **Sourcing Corpora**

1. There are several freely available sources already collected and curated
2. You can scrape the web to source documents using **Beautiful Soup** or **rvest.**

* **What sources of text are available to me?**

웹사이트의 데이터를 가져오는 것이 일반적이다. 웹사이트의 text source를 가져오는 것에 허용범위가 있는데, “robot.txt”로 검색로봇에게 사이트 및 웹 페이지를 수집할 수 있도록 허용하거나 제한하는 국제 권고안이 있다. robot.txt의 구조 “allow”, “disallow”는 허용범위를 나타내며, 허용한 범위내에서 데이터를 가져오는 것이 내가 주어진 유효한 데이터가 될 것이다.

* **What types of questions can I answer with text data?**

Text 데이터는 비정형 데이터로 자연어 처리 기술과 문서 처리 기술을 적용하여 유용한 정보를 추출, 가공하는 목적으로 사용될 수 있다. 즉, 감정분석을 통해 경향성을 판단하기에 좋은 도구가 될 것이다.

* **Beautiful Soup and rvest**

**Beautiful Soup**

*"""  
1.requests를 이용하여 웹에 있는 소스를 가져올 수 있다.  
2.requests는 html을 Python이 이해하는 객체구조로 만들 수 없기에  
 beutifulsoup을 이용하여 html코드를 Python이 이해하는 객체 구조로 변경할 수 있다.  
 이를 통해 의미있는 text 정보를 추출해 낼 수 있다.*

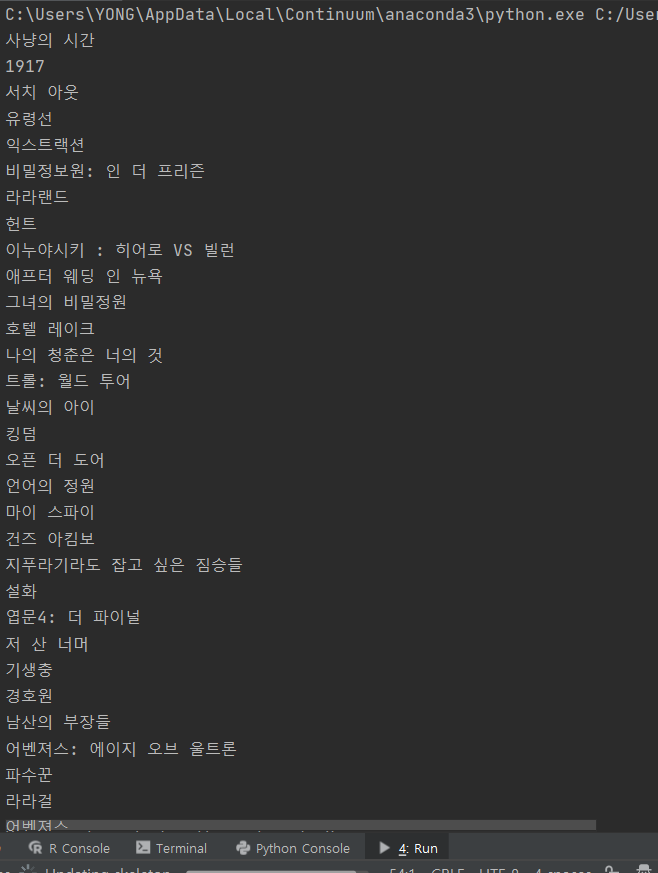
*Beutifulsoup를 install 하지 않았다면*

*pip install beautifulsoup4*

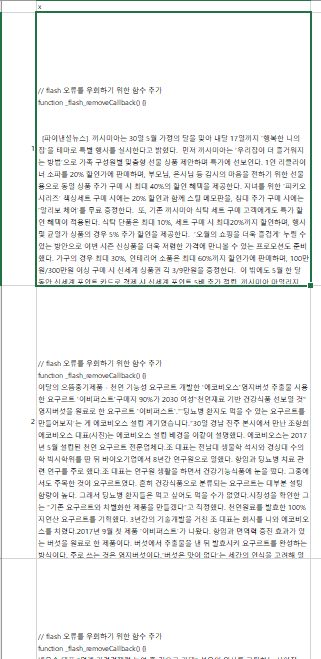
*코드를 통해 설치하도록 한다.  
"""*import requests  
from bs4 import BeautifulSoup  
  
req = requests.get("https://movie.naver.com/movie/sdb/rank/rmovie.nhn")  
html = req.text  
  
soup = BeautifulSoup(html, 'html.parser')  
movie\_titles = soup.select(  
 'tr > td.title'  
 )  
  
"""  
태그 확인을 통해 제목(text)만 추출하는 과정을 확인할 수 있음  
clip1 = movie\_titles[0]  
clip1\_title = clip1.select\_one('td.title a')  
clip1\_real = clip1\_title.text  
print(clip1\_real)  
"""  
  
for info in movie\_titles:  
 title = info.select\_one('td.title a').text  
 print(title)  
  
"""  
네이버영화 웹사이트로부터 1위부터 50위까지의 영화제목을 크롤링하는 과정에서  
requests와 Beutifulsoup을 활용하는 방법을 확인할 수 있었음.

"""

#결과물



**rvest**

#R 환경에서의 크롤링은 rvest package를 이용할 수 있다.  
#뉴스 기사내용을 추출하는 과정으로 rvest를 활용해본다.  
  
*library*(rvest)  
#library(dplyr)  
  
#<-기호는 어떠한 변수에 값을 넣으라는 뜻이다.  
#<- NULL은 empty space를 뜻한다.  
base\_url <- 'https://news.naver.com/main/list.nhn?mode=LS2D&mid=shm&sid2=771&sid1=101&date=20200430&page='  
urls <- NULL  
  
for(x in 1:2){  
 urls[x] <- *paste0*(base\_url, x)  
 #paste0(,) 띄어쓰기 없이 문장을 붙여준다.  
}  
  
#html1 <- read\_html(urls[1]) #url[1]의 html을 변수 html1에 저장  
#html2 <- html\_nodes(html1, '.content') #html node 중 content 클래스 하위 모두 html2에 저장  
#html3 <- html\_nodes(html2, 'a') #실질적으로 필요한 a로 끝나는 부분을 추출하는 과정을 html3에 저장  
  
#links <- html\_attr(html3,'href') #링크만 남기는 작업을 하기위해 herf의 주소만을 가져옴  
#위와 같은 긴 코드를 간략하게 만들어주는 pipe  
#install.packages('dplyr') 코드를 통해 설치해주고 라이브러리로 불러줌  
#links <- html1 %>% html\_nodes('.content') %>% html\_nods('a') %>% html\_attr('href')  
#위와 같이 간략히 작성할 수 있음  
  
#제목에만 링크가 있는 것이아니라 이미지를 통해 접근이 가능하므로 중복된 링크를 제거해주는 기능을 사용해야함  
#links <- unique(links)  
#불필요한 자료가 함께 들어가 있다면 grep("string", links)를 통해 제거 해줌  
#links[-grep("string",links)]를 입력하도록 함.  
  
links <- NULL  
for(url in urls){  
 html <- *read\_html*(url)  
 links <- *c*(links, html1 %>% *html\_nodes*('.content') %>% *html\_nodes*('a') %>% *html\_attr*('href') %>% *unique*())  
#c(,)는 links라는 변수에 자료를 계속 더하라는 뜻임  
}  
links <- links[*grep*("http", links)]  
  
txts <- NULL  
for(link in links){  
 html <- *read\_html*(link)  
 txts <- *c*(txts, html %>% *html\_nodes*('#articleBodyContents') %>% *html\_text*())  
}  
#결과물  
*write.csv*(txts, "text.csv")

* **Text Preprocessing**
* **What are the choices I can make?**

Preprocessing은 데이터 전처리 과정을 의미한다. 데이터 전처리는 데이터 내에서 불필요한 부분을 제거 또는 수정하여 최대한 데이터 소스의 정확성을 높이는 것이다. Text Preprocessing의 경우는 토큰화, 정제와 정규화, 어간 추출, 불용어 등이 전처리 과정으로 선택될 수 있다.

* **토큰화(Tokenizaion)**

크롤링을 통해 얻어낸 데이터를 필요에 맞게 다듬기 위해 토큰이라 불리는 단위로 나누는 작업인 토큰화를 진행해야 한다. 토큰의 단위는 상황에 따라 다르다. 토큰의 기준을 단어로 하는 경우가 있다. 예를 들어 구두점(온점, 콤마, 물음표, 세미콜론, 느낌표 등)을 기준으로 토큰화를 진행하면

|  |  |
| --- | --- |
| 토큰화 전 | Time is an illusion. Lunchtime double so! |
| 토큰화 후 | Time, is, an, illusion, Lunchtime, double, so |

간단한 토큰화를 진행했을 때, 발생하는 문제가 있다. 토큰화의 기준이 구두점만으로 충족되지 않는 경우가 생기기에 기준을 잘 선택해야한다. 다음은 간단한 코드를 통해 표준 토큰화를 어떻게 진행할 수 있는지 알아보고자 한다.

*"""  
표준으로 쓰이고 있는 토큰화 방법 중 하나인 Penn Treebank Tokenizaion의 규칙으로 코딩한다.  
규칙1. 하이푼으로 구성된 단어는 하나로 유지한다.  
규칙2. Dosen't와 같이 아포스트로피가 함께하는 단어는 분리한다.  
"""*from nltk.tokenize import TreebankWordTokenizer  
tokenizer = TreebankWordTokenizer()  
text = "Starting a home-based restaurant may be an ideal. it doesn't have a food chain or restaurant of their own."  
print(tokenizer.tokenize(text))

>> ['Starting', 'a', 'home-based', 'restaurant', 'may', 'be', 'an', 'ideal.', 'it', 'does', "n't", 'have', 'a', 'food', 'chain', 'or', 'restaurant', 'of', 'their', 'own', '.']

다음은 문장 토큰화이다. 경우에 따라 문장 단위로 구분하는 문장 토큰화가 필요할 수 있다.

from nltk.tokenize import sent\_tokenize  
text = "His barber kept his word. But keeping such a huge secret to himself was driving him crazy. Finally, the barber went up a mountain and almost to the edge of a cliff. He dug a hole in the midst of some reeds. He looked about, to mae sure no one was near."  
print(sent\_tokenize(text))

>>> ['His barber kept his word.', 'But keeping such a huge secret to himself was driving him crazy.', 'Finally, the barber went up a mountain and almost to the edge of a cliff.', 'He dug a hole in the midst of some reeds.', 'He looked about, to mae sure no one was near.']

* **정제(Cleaning), 정규화(Normalization)**

토큰화 전후로 정제를 통해 가지고 있는 데이터로부터 노이즈 데이터를 제거하고, 정규화를 통해 표현 방법이 다른 단어들을 통합시켜서 같은 단어로 만들어주는 과정을 거친다.

규칙에 기반하여 표기가 다른 단어들을 통합한다. 예를 들어, USA와 US는 같은 의미를 지니고 있으므로 하나의 단어로 정규화 할 수 있다. 뿐만 아니라 대, 소문자의 통합은 단어의 개수를 줄일 수 있는 또 다른 정규화 방법이다.

등장 빈도가 작거나 길이가 짧은 단어를 삭제하는 것만으로도 좋은 정제, 정규화 효과를 지닌다.

import re  
text = "I was wondering if anyone out there could enlighten me on this car."  
shortword = re.compile(r'\W\*\b\w{1,2}\b')  
print(shortword.sub('', text))

was wondering anyone out there could enlighten this car.

노이즈 데이터의 특징을 잡아낼 수 있다면, 정규 표현식을 통해서 이를 제거할 수 있는 경우가 많다. 계속 등장하는 글자들을 규칙에 기반하여 한 번에 제거하는 방식으로 유용하게 쓰일 것이다.

* **어간 추출(Stremming)과 표제어 추출(Lemmatization)**

어간 추출과 표제어 추출 모두 문서 내의 단어 수를 줄이는 것을 의미한다. 단어의 빈도수를 기반으로 문제를 풀고자 할 때, 사용하는 자연어 처리 문제에 자주 등장한다.

우선 표제어 추출은 단어들이 다른 형태를 가지더라도, 기반이 되는 단어를 찾아가서 단어의 개수를 줄일 수 있는지 판단한다. 예를 들어 be동사의 am, are, is는 be동사로 볼 수 있기에, 이 들은 표제어 be로 판단될 수 있다.

표제어 추출을 하는 가장 섬세한 방법은 단어의 형태학적 파싱을 진행하는 것인데, 어간과 접사를 구분하여 분리하는 작업을 형태학적 파싱이라고 한다. 예를 들어 cats의 경우 어간은 cat이며, 접사는 -s가 된다.

from nltk.stem import WordNetLemmatizer  
n=WordNetLemmatizer()  
words=['policy', 'doing', 'organization', 'have', 'going', 'love', 'lives', 'fly', 'dies', 'watched', 'has', 'starting']  
print([n.lemmatize(w) for w in words])

>>> ['policy', 'doing', 'organization', 'have', 'going', 'love', 'life', 'fly', 'dy', 'watched', 'ha', 'starting']

표제어 추출은 어간 추출과 달리 단어의 형태가 적절히 보존되는 양상을 보이는 특징을 지니고 있다.

어간 추출은 형태학적 분석을 단순화한 버전이다. 정해진 규칙만 짐작하여 단어의 어미를 자르는 작업이다. 따라서 어간 추출 후에 나오는 결과는 사전에 존재하지 않을 수 있다.

from nltk.stem import PorterStemmer  
from nltk.tokenize import word\_tokenize  
s = PorterStemmer  
text = "This was not the map we found in Billy Bones's chest, but an accurate copy, complete in all things--names and heights and soundings--with the single exception of the red crosses and the written notes."  
words = word\_tokenize(text)  
print(words)

>>> ['This', 'was', 'not', 'the', 'map', 'we', 'found', 'in', 'Billy', 'Bones', "'s", 'chest', ',', 'but', 'an', 'accurate', 'copy', ',', 'complete', 'in', 'all', 'things', '--', 'names', 'and', 'heights', 'and', 'soundings', '--', 'with', 'the', 'single', 'exception', 'of', 'the', 'red', 'crosses', 'and', 'the', 'written', 'notes', '.']

##포터 알고리즘의 어간 추출의 규칙은 다음과 같다.

##ALIZE > AL

##ANCE > 제거

## ICAL > IC

* **불용어(Stopword)**

현재 가지고 있는 데이터에서 유의미한 단어 토큰만을 선별하기위해 의미 없는 단어 토큰을 제거하는 작업도 필요하다. 의미가 없다는 것은 자주 등장하기도 하지만, 분석을 하는 것에 있어 큰 도움이 되지 않는 단어들을 말한다. 다음은 불용어를 제거한 Text의 예시를 코드로 표현한 것이다.

from nltk.corpus import stopwords  
from nltk.tokenize import word\_tokenize  
  
example = "Familt is not an important thing. It's everything"  
stop\_words = set(stopwords.words('english'))  
  
word\_tokens = word\_tokenize(example)  
result = []  
for w in word\_tokens:  
 if w not in stop\_words:  
 result.append(w)  
  
print(word\_tokens)  
print(result)

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

['Familt', 'important', 'thing', '.', 'It', "'s", 'everything']

* **What is winsorizing?**

통계 분포는 평균에서 크게 벗어난 값으로 인해 많은 정규성에 많은 영향을 받을 수 있다. 따라서 이러한 값들을 백분위 수로 임의대로 지정하는 것이다. 예를 들어 5% 백분위 이하는 모두 5% 백분위수와 동일하게 두고, 95% 백분위 이상은 모두 95% 백분위와 동일하게 둔다.

Winsorizing은 크게 벗어난 값을 가진 데이터를 다루는데 좋은 효과를 볼 수 있다.

import scipy.stats  
import numpy as np  
  
a = np.array([92, 19, 101, 58, 1053, 91, 26, 78, 10, 13, -40, 101, 86, 85, 15, 89, 89, 28, -5, 41])  
b = scipy.stats.mstats.winsorize(a, limits=[0.05, 0.05])  
print(b)  
  
"""  
limits=[0.05, 0.05]로 설정하여 winsorizing을 했을 때,  
5%에 해당하는 수는 5%에 가장 근접한 값으로 변하며,  
95%에 해당하는 수는 95%에 가장 근접한 값으로 바뀌어 산출되는 모습을 볼 수 있다.  
"""

>>> [ 92 19 101 58 101 91 26 78 10 13 -5 101 86 85 15 89 89 28

-5 41]

* **Descriptive Inference**
* **What’s a DTM**

**문서 단어 행렬 Document-Term Matrix(DTM)**은 여러 문서들의 집단에서 단어를 추출한 뒤 행에 출현했던 단어 리스트를 나열하고, 열에 각 문서들을 나열한 행렬이다. 즉, 다수의 문서에서 등장하는 각 단어들의 빈도를 행렬로 표현한 것을 말한다.

하지만 Document-Term Matrix에는 한계점이 존재한다. **첫번째는** **희소표현**이다. 단어 집합의 크기는 벡터의 차원이 된다. 만약 가지고 있는 데이터가 너무 방대한 데이터라면 문서 벡터의 차원은 수백만 이상의 차원이 될 수 있다. 또한 특정 단어의 빈도수가 없는 벡터 공간에서는 대부분의 값이 0으로 할당된 것은 공간 낭비를 야기시킨다. 대부분의 값이 0인 표현을 희소벡터 또는 희소 행렬이라고 부르고, 희소 벡터는 많은 양의 저장 공간과 계산을 위한 리소스를 필요로 한다.

**두번째 한계점은** **단순 빈도 수 기반 접근**이라는 것이다. 예를 들어 영어를 이용한 DTM을 제작했을 때 불용어인 ‘The’ 와 ‘a’ 등이 자주 발생할 수 있다. 그런데 분석하고자 하는 문서에서 위와 같은 불용어가 많이 나왔다고 유사한 문서들이라고 판단할 수 없다. 즉, 여러 문서에 등장하는 모든 단어에 대해서 빈도 표기를 하는 방법은 때로는 한계를 가지게 된다. 중요단어와 불필요 단어의 가중치를 주는 방식을 TF-IDF라고한다.

**단어빈도와 역 문서 빈도** **TF-IDF(Term Frequency-Inverse Document Frequency)**는 단어의 빈도와 역 문서빈도(문서의 빈도에 특정 식을 취함)를 사용하여 DTM 내의 각 단어들마다 중요한 정도를 측정하는 방법이다. 사용방법은 우선 DTM 내의 각 단어들마다 중요한 정도를 가중치로 주는 방법을 사용한다.

문서를 d, 단어를 t, 문서의 총 개수를 n이라고 할 때

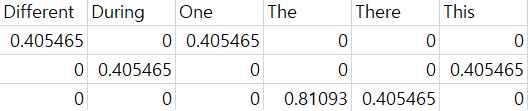
|  |  |
| --- | --- |
| **TF** | 특정 문서 d에서의 특정 단어 t의 개수 |
| **DF** | 특정 단어 t가 등장한 문서의 수 |
| **IDF**  idf(d,t)=log(n/1+df(t)) | DF에 반비례하는 수  (문서의 개수 n이 상당히 큰 수라면 IDF값이 기하급수적으로 커지게 되므로 log를 사용) |

TF-IDF는 이들을 곱하여 수치적으로 표현할 수 있다. 예를 들어 문서1에서는 ‘교수님’을 한 번 언급했고, 문서2에서는 ‘교수님’을 두 번 언급했다면 문서2에서 ‘교수님’의 중요성이 문서1보다 크다고 판단한다.

다음은 TF-IDF를 적용, Python을 이용해 DTM을 구현해보고자 한다.

import pandas as pd # 데이터 프레임의 사용을 위해 'pandas'이용  
 # 표와 같은 틀을 제공한다고 생각하자.  
from math import log # 가중치 계산을 위한 log를 사용할 수 있도록 한다.  
  
docs = ['Base oils are used to manufacture products including lubricating greases, motor oil and metal processing fluids. Different products require different compositions and properties in the oil. One of the most important factors is the liquid’s viscosity at various temperatures. Whether or not a crude oil is suitable to be made into a base oil is determined by the concentration of base oil molecules as well as how easily these can be extracted.',  
 'Base oil is produced by means of refining crude oil. This means that crude oil is heated in order that various distillates can be separated from one another. During the heating process, light and heavy hydrocarbons are separated – the light ones can be refined to make petrol and other fuels, while the heavier ones are suitable for bitumen and base oils',  
 'There are large numbers of crude oils all around the world that are used to produce base oils. The most common one is a type of paraffinic crude oil, although there are also naphthenic crude oils that create products with better solubility and very good properties at low temperatures. By using hydrogenation technology, in which sulfur and aromatics are removed using hydrogen under high pressure, you can obtain extremely pure base oils, which are suitable when quality requirements are particularly stringent.']  
  
vocab = list(set(w for doc in docs for w in doc.split()))  
vocab.sort() #sort()는 오름차순 정렬을 뜻한다.  
  
N = len(docs) #문서의 수  
#아래 코드는 TF와 IDF를 구현하는 함수  
def tf(t, d):  
 return d.count(t)  
  
def idf(t):  
 df = 0  
 for doc in docs:  
 df += t in doc  
 return log(N/(df + 1))  
  
def tfidf(t, d):  
 return tf(t, d) \* idf(t)  
  
#DTM을 데이터 프레임에 저장하여 출력  
result = []  
for i in range(N):  
 result.append([])  
 d = docs[i]  
 for j in range(len(vocab)):  
 t = vocab[j]  
 result[-1].append(tf(t, d))  
  
tf\_ = pd.DataFrame(result, columns = vocab)  
  
#idf를 데이터 프레임에 저장하여 출력  
result = []  
for j in range(len(vocab)):  
 t = vocab[j]  
 result.append(idf(t))  
  
idf\_ = pd.DataFrame(result, index= vocab, columns= ["IDF"])  
  
#각 idf가 문서 1~3에 어떠한 비중으로 자리하고 있는지 확인  
result = []  
for i in range(N):  
 result.append([])  
 d = docs[i]  
 for j in range(len(vocab)):  
 t = vocab[j]  
  
 result[-1].append(tfidf(t, d))  
  
tfidf\_ = pd.DataFrame(result, columns=vocab)  
tfidf\_.to\_csv(('dtm.csv'))

결과물의 일부

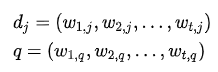
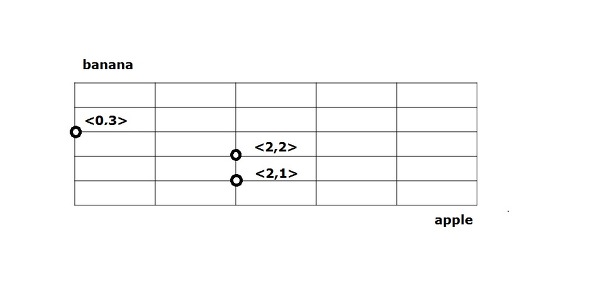


* **Lexical Diversity Measures**

Lexical diversity는 총 고유 단어의 개수에 대한 어떠한 고유 단어의 비율을 나타내며, Lexical diversity measures는 어휘 밀도를 측정하는 것을 의미한다. 어휘의 밀도를 측정하기 위해서 다음과 같은 요소들을 고려하여 측정하는 것을 제시한다.

|  |  |
| --- | --- |
| **특성** | **측정법** |
| Variability  (다양성) | Measure of Textual Lexical Diversity(MTLD)  문자의 어휘가 얼마나 다양성을 가지고 있는가에 대한 분석틀 |
| Volume  (크기) | Total number of words in the text  문자 속 전체 단어들의 수 |
| Evenness  (균일) | Standard deviation of tokens per type  유형별 표준편차 |
| Rarity  (희소) | Mean BNC rank  BNC는 영국의 영어 text corpus |
| Dispersion  (분산) | Mean distance between tokens of type  유형별 분산 확인 |
| Disparity  (상이) | Mean number of words per sense or Latent Semantic Analysis  문서 및 용어와 관련된 개념을 생성, 문서에 포함된 용어 간의 관계를 분석하는 것을 의미 |

* **Vector space model**
  + **What is the VSM useful for?**

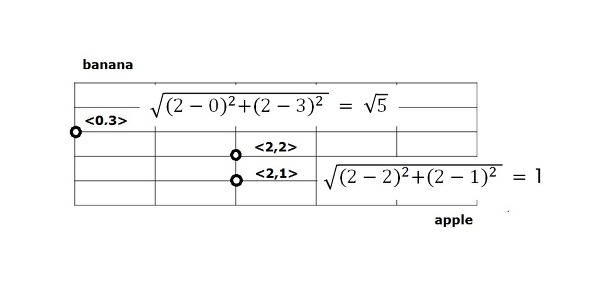
벡터 공간 모델은 텍스트 문서를 단어, 문장 등이 식별자로 구성된 벡터로 표현하는 대수적 모델이다. 정보 검색, 정보 필터링 및 검색 엔진의 색인이나 연관도 순위에 사용된다. (d = 문서 벡터(문서가 가진 단어 또는 문장), q = 질의 벡터(해당하는 단어 또는 문장을 가지고 있는가?))

예를 들어 세상에 단어가 딱 2개 밖에 없다면(apple, banana) 문서1은 banana만 3개 있다고 했을 때, (0,3)에 해당하는 벡터 공간 위에 뿌려진다.

각각의 차원은 개별 단어에 대응된다. 어떤 단어가 문서에 포함되면, 해당 단어는 0이 아닌 벡터 값을 갖는다. 단어 가중치라고도 불리는 이 값을 산출하는 방법에는 여러가지가 있다. 가장 널리 알려진 방식이 DTM을 구현하며 배운 TF-IDF 방식이다.

벡터 공간 모델에서 단어의 의미는 그 적용 대상에 따라 달라진다. 일반적으로는 하나의 단어나 키워드, 또는 문장을 의미한다. 벡터 차원의 크기는 데이터에 포함된 단어의 개수와 같다. 벡터 연산은 문서들 간의 유사성을 비교하는 데에도 사용될 수 있다.

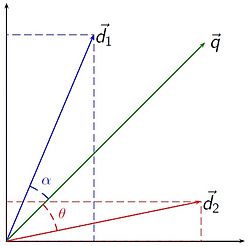
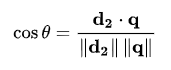
* + **Euclidean Distance**

유클리디언 거리는 각 문서의 문서 벡터와 질의 벡터의 거리를 계산하는 방법이다. 계산 방법은 벡터의 내적을 구하는 방법과 동일하다.

위 그림과 같이 <2,2>로부터 유클리디언 거리를 구하면 <2,1>의 위치에 있는 문서의 거리가 짧으므로 유사도가 높다고 할 수 있다.

하지만 문서의 방향성을 고려하지 않고 같은 단어의 중복으로 이루어진 문서의 경우는 질의 벡터에 따른 거리가 멀어져 유사도는 높지만 선택 시, 고려되지 않을 가능성이 높아진다.

* + **Cosine Distance**

 각도를 고려한 유사도 계산법이다. 키워드 검색에서 사용되는 문서의 연관도는 **문서 유사도 이론**으로 계산될 수 있다. **문서 유사도 이론에서 문서 유사도는 문서 벡터 간의 각도의 편차를 이용하여 산출**된다. 실제 적용시, **문서 벡터 간의 각도 자체보다는 다음과 같이 각도의 코사인 값을 이용**하는 것이 더 간단하다.

분자는 문서 벡터(d2)와 질의 벡터(q)의 교차점(내적)에 해당하며, 각 벡터의 크기(norm)을 구할 수 있다. **벡터의 모든 요소는 음수가 아닌 값으로, 코사인 값이 0인 경우 질의 벡터와 문서 벡터가 직교하며 겹치는 부분이 전혀 없음(서로 동일하게 포함하고 있는 단어가 하나도 없음)을 의미한다. 문서 벡터의 길이와 방향성을 함께 고려할 수 있다는 점이 장점이다.** (즉, 코사인 값이 1인 경우 질의 벡터에 따른 유사도가 굉장히 높은 문서 벡터로 판단할 수 있다.)

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))  
  
"""  
문서1 : 저는 사과 좋아요  
문서2 : 저는 바나나 좋아요  
문서3 : 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  
"""  
  
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)) #문서2와 문서3의 코사인 유사도  
  
#result  
#0.6666666666666667  
#0.6666666666666667  
#1.0000000000000002

문서2와 3을 비교했을 때 단순히 빈도수의 증가만 이루어졌음을 알 수 있다. 문서의 방향성을 고려할 때, VSM의 거리와 상관없이 방향성을 찾기에 좋은 분석 Tool을 제공해준다는 것을 확인할 수 있었다.

* + **What is useful about TF-IDF**

우리는 이미 DTM을 공부하며, DTM이 단순 단어 빈도수 비교라는 단점을 보완하기 위한 방법으로 TF-IDF를 사용했다. 복습의 의미로 한 번 더 학습한다면, **TF-IDF는 단어의 빈도와 역 문서 빈도**(문서의 빈도에 특정 식을 취함)를 사용하여 DTM 내의 각 단어들마다 **중요한 정도를 가중치 주는 방법**을 의미한다. TF-IDF는 주로 문서의 유사도를 구하는 작업, 검색 시스템에서 검색 결과의 중요도를 정하는 작업, 문서 내에서 특정 단어의 중요도를 구하는 작업 등에 쓰일 수 있다.

지난 시간에는 파이썬으로 직접 TF-IDF를 구현했지만, 이번 시간에는 사이킷런을 이용하여 DTM과 TF-IDF를 만드는 연습을 하고자 한다.

from sklearn.feature\_extraction.text import CountVectorizer  
#DTM 만들기  
  
corpus = [  
 'you know I want your love',  
 'I like you',  
 'what should I do ',  
]  
vector = CountVectorizer()  
print(vector.fit\_transform(corpus).toarray())  
# 코퍼스로부터 각 단어의 빈도 수를 기록  
print(vector.vocabulary\_)  
# 각 단어의 인덱스가 어떻게 부여되었는지를 보여줌

[[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, 'love': 3, 'like': 2, 'what': 6, 'should': 4, 'do': 0}

>> 해당하는 단어 옆의 숫자는 인덱스를 나타낸다. 인덱스는 열의 번호라고 생각하면 된다.

from sklearn.feature\_extraction.text import TfidfVectorizer  
#TF-IDF 실습, TF-IDF를 자동 계산해주는 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\_)

C:\Users\YONG\AppData\Local\Continuum\anaconda3\python.exe C:/Users/YONG/PycharmProjects/textmining1/venv/Lib/site-packages/easy\_install.py

[[0. 0.46735098 0. 0.46735098 0. 0.46735098

0. 0.35543247 0.46735098]

[0. 0. 0.79596054 0. 0. 0.

0. 0.60534851 0. ]

[0.57735027 0. 0. 0. 0.57735027 0.

0.57735027 0. 0. ]]

{'you': 7, 'know': 1, 'want': 5, 'your': 8, 'love': 3, 'like': 2, 'what': 6, 'should': 4, 'do': 0}

사이킷런은 TF-IDF를 자동 계산해주는 TfidVectorizer를 제공하는데, 우리가 윗 장에서 배웠던 가중치를 적용하는 식과 조금은 상이하다. 즉, 보편적인 TF-IDF 식에서 좀 더 조정된 다른 식을 사용한다. 하지만 여전히 TF-IDF가 가진 의도를 그대로 갖고 있으므로 사이킷런의 TF-IDF를 그대로 사용해도 괜찮다.

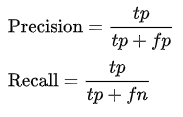
* **Supervised Learning**
  + **Supervised Learning: What is it?**

Supervised learning은 지도학습을 뜻한다. 지도학습은 **훈련 데이터로부터 하나의 함수를 유추해내기 위한 기계학습의 한 방법**이다. 훈련 데이터는 일반적으로 입력 객체에 대한 속성을 벡터 형태로 포함하고 있으며 각각의 벡터에 대해 원하는 결과가 무엇인지 표시되어 있다. 이렇게 유추된 **함수 중 연속적인 값을 출력하는 것을 회귀분석**이라고 하고 주어진 입력 **벡터가 어떤 종류의 값인지 표현하는 것을 분류라고 한다.**

훈련 데이터로부터 하나의 함수가 유추되고 나면 해당 함수에 대한 평가를 통해 파라미터(범위)를 최적화한다. 이러한 평가를 위해 교차검증이 이용되며 이를 위해 검증집합을 다음의 3가지로 나눈다

* 훈련 집합
* 검증 집합
* 테스트 집합

이러한 교차검증을 통해 훈련된 각 함수에 대한 정밀도(Precision)와 재현정도(Recall)를 측정할 수 있다.

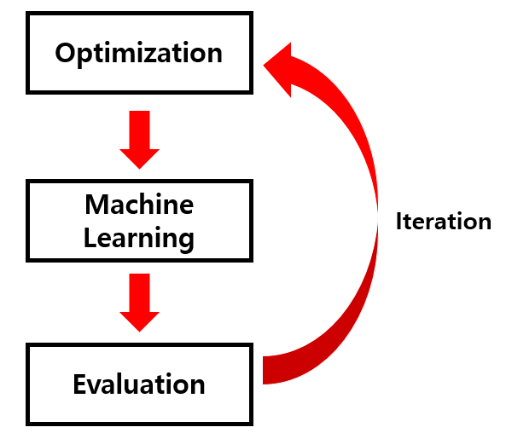


|  |  |  |  |
| --- | --- | --- | --- |
|  | | 실제결과 / 분류 | |
| 참 | 거짓 |
| 추론된  결과 / 분류 | 참 | Tp  True positive | Fp  False positive |
| 거짓 | Fn  False negative | Tn  True negative |

* + **Feature Selection: What can I use?**

Feature selection은 모델을 구성하기 위한 변수를 선택하는 과정이다. Feature가 너무 많으면 overfitting(실 데이터에 대한 오류가 증가하는 현상)의 위험이 있고, 모델이 이상적인 방향으로 학습하지 않을 위험이 있기 때문에 이상적인 변수 조합을 찾아야한다.

Feature selection을 사용하면 사용자가 더 해석하기 쉽게 모델을 단순화할 수 있고 훈련 시간을 축소시킬 수 있다. Feature selection은 대표적으로 Wrapper method가 있다.

* **Wrapper method**

우선 Feature의 조합을 정하고, 기계학습을 돌리고, 성능을 평가하는 과정을 반복한다. 조합을 반복적으로 바꿔가며 실행하는데 이 과정에서 가장 성능이 좋은 조합을 찾아 낼 수 있다.

Wrapper method는 굉장히 많은 시간이 필요하고, 훈련 데이터셋에 너무 overfitting 될 수 있다.

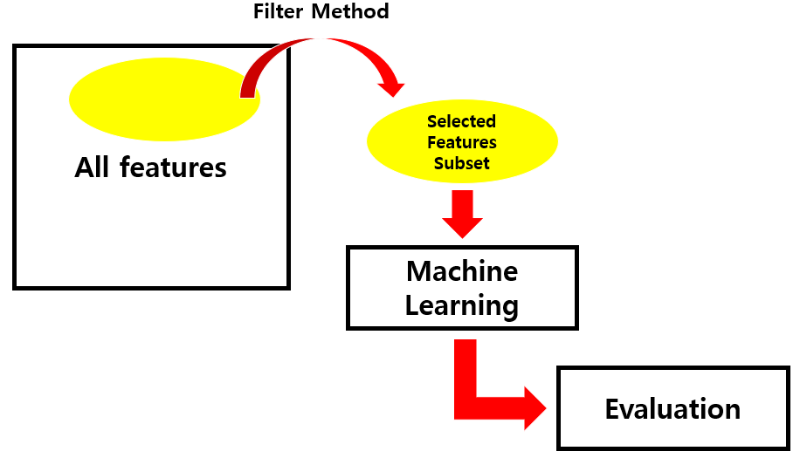
Wrapper method에는 3가지 종류가 있다. (step forward, step backwards, exhaustive) 그 중 step forward feature selection을 파이썬으로 구현해보고자 한다.

import pandas as pd  
import numpy as np  
from sklearn.model\_selection import train\_test\_split  
from sklearn.feature\_selection import VarianceThreshold  
  
#데이터 전처리 과정 > 데이터는 kaggle의 공유 데이터를 이용했다.  
#---------------------------------------------------------------------------------------------------------------------  
paribas\_data = pd.read\_csv("train.csv", nrows=20000)  
paribas\_data.shape #array의 크기를 확인할 수 있다 (행=20000, 열=132)  
#print(paribas\_data.shape)  
  
num\_colums = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']  
#num\_colums를 해당 코드와 같은 array로 지정  
#int16은 정수의 ~까지를 포함하는수  
#int16, int32와 같이 type을 설정해주는 경우로 해석할 수 있다.  
  
numerical\_columns = list(paribas\_data.select\_dtypes(include=num\_colums).columns)  
#print(numerical\_columns)  
#위 코드는 숫자로 구성된 columns만을 추출하는 작업으로 paribas\_data에서 num\_colunms에 해당하는 columns를 찾아 반환한다.  
  
paribas\_data = paribas\_data[numerical\_columns]  
#paribas\_data를 paribas\_data의 numerical\_columns에 해당하는 값만으로 다시 저장  
paribas\_data.shape #array의 크기를 확인할 수 있다 (행=20000, 열=113) > 초기 값에서 열의 개수가 줄어들었음을 확인할 수 있다.  
#print(paribas\_data)  
  
train\_features, test\_features, train\_labels, test\_labels = train\_test\_split(  
 paribas\_data.drop(labels=['ID', 'target'], axis=1),  
 paribas\_data['target'],  
 test\_size=0.2,  
 random\_state=41  
)  
  
correlated\_features = set() #set()은 집합을 의미하는 함수이다.  
correlation\_matrix = paribas\_data.corr() #corr()은 상관관계를 분석해주는 함수이다.  
  
for i in range(len(correlation\_matrix .columns)): #상관관계 분석 matrix의 열 수 만큼 반복하여  
 for j in range(i):  
 if abs(correlation\_matrix.iloc[i, j]) > 0.8: #각 인덱스에 해당하는 값의 절댓값이 0.8보다 크면  
 colname = correlation\_matrix.columns[i] #colname에 i번째 colunm이름을  
 correlated\_features.add(colname) # correlated\_features.add(colname) > correlated\_features에 colname에 해당하는 값 저장  
  
train\_features.drop(labels=correlated\_features, axis=1, inplace=True)  
test\_features.drop(labels=correlated\_features, axis=1, inplace=True)  
  
train\_features.shape, test\_features.shape  
#---------------------------------------------------------------------------------------------------------------------  
  
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier  
from sklearn.metrics import roc\_auc\_score  
  
from mlxtend.feature\_selection import SequentialFeatureSelector  
  
feature\_selector = SequentialFeatureSelector(RandomForestClassifier(n\_jobs=-1),  
 k\_features=15,  
 forward=True,  
 verbose=2,  
 scoring='roc\_auc',  
 cv=4)  
  
features = feature\_selector.fit(np.array(train\_features.fillna(0)), train\_labels)  
filtered\_features= train\_features.columns[list(features.k\_feature\_idx\_)]  
filtered\_features  
  
clf = RandomForestClassifier(n\_estimators=100, random\_state=41, max\_depth=3)  
clf.fit(train\_features[filtered\_features].fillna(0), train\_labels)  
  
train\_pred = clf.predict\_proba(train\_features[filtered\_features].fillna(0))  
print('Accuracy on training set: {}'.format(roc\_auc\_score(train\_labels, train\_pred[:,1])))  
  
test\_pred = clf.predict\_proba(test\_features[filtered\_features].fillna(0))  
print('Accuracy on test set: {}'.format(roc\_auc\_score(test\_labels, test\_pred [:,1])))

Accuracy on training set: 0.7072327148174093

Accuracy on test set: 0.7096973252804142

* **Filter Method**



Filter Method는 전처리 과정에서 미리 feature selection을 통계적 방법으로 실행하고, 모델을 찾는 방법이다. 가장 많이 사용되는 통계적 방법으로 종속변수와 독립변수 간의 피어슨 상관계수를 이용하는 것이다.

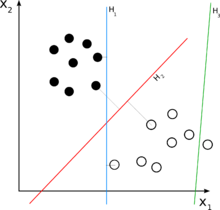
피어슨 상관계수는 두 변수 X, Y간의 선형 상관 관계를 계랑화 한 수치를 나타낸다. 또한 피어슨 상관 계수는 코시-슈바르츠 부등식에 의해 +1, -1 사이의 값을 가지며, +1은 완벽한 양의 선형 상관 관계, 0은 선형 상관 관계없음, -1은 완벽한 음의 선형 상관 관계를 의미한다.

* + **Logistic Regression: When should I use?**

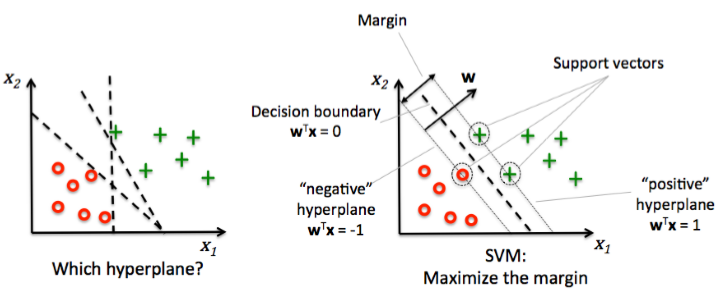
로지스틱 회귀의 목적은 **일반적인 회귀 분석의 목표와 동일하게 종속 변수와 독립 변수 간의 관계를 구체적인 함수로 나타내어 향후 예측 모델에 사용**하는 것이다. 이는 독립 변수의 선형 결합으로 종속 변수를 설명한다는 관점에서는 선형 회귀 분석과 유사 하지만 로지스틱 회귀는 선형 회귀 분석과는 다르게 종속 **변수가 범주형 데이터를 대상으로 하며 입력 데이터가 주어졌을 때 해당 데이터의 결과가 특정 분류로 나뉘기 때문에 일종의 분류 기법**으로도 볼 수 있다.

* + **Support Vector Machine: When should I use?**

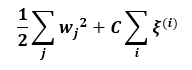
**패턴 인식과 자료 분석을 위한 지도 학습 모델이며 주로 분류와 회귀 분석을 위해 사용**한다. 두 카테고리 중 어느 하나에 속한 데이터의 집합이 주어졌을 때, SVM 알고리즘은 주어진 데이터 집합을 바탕으로 하여 새로운 데이터가 어느 카테고리에 속할지 판단하는 비확률적 이진 선형 분류 모델을 만든다. 만들어진 분류 모델은 데이터가 사상된 공간에서 경계로 표현되며 SVM알고리즘은 그 중 가장 큰 폭을 가진 경계를 찾는 알고리즘이다.

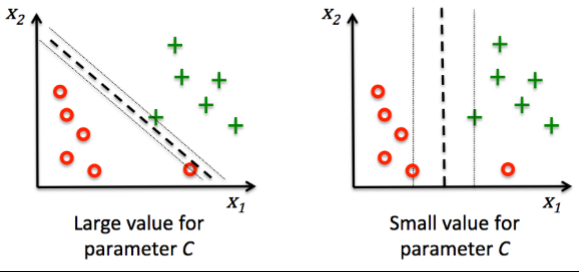
주어진 데이터 점들이 두 개의 클래스 안에 각각 속해 있다고 가정했을 때, 새로운 데이터 점이 두 클래스 중 어느 곳에 속하는지 결정하는 것이 목표이다. 서포트 벡터 머신에서, 데이터 점이 P-차원의 벡터로 주어졌을 때, 이러한 데이터 점을 (p-1)차원의 초평면으로 분류할 수 있는지를 확인한다. 이러한 작업을 선형 분류라고 한다.

위 그림에서 H3는 두 클래스의 점들을 제대로 분류하지 못한다. H1과 H2는 두 클래스의 점들을 분류하는데, H2가 정확한 분류를 하게 될 것을 예상할 수 있다.

SVM은 margin을 최대가 되도록 하는 알고리즘이다. 여기서 말하는 margin은 분류를 위한 경계선과 이 경계선에 가장 가까운 트레이닝 데이터 사이의 거리를 말한다. 이 경계선에 가장 가까운 트레이닝 데이터들을 support vector라고 한다.

빨간색 원과 초록색 더하기 기호로 표시된 집단을 분류하는 경계는 다양하게 존재할 수 있는데, SVM은 두 집단을 분류하는 경계선 중 support vector와의 거리가 가장 멀리 떨어져 있는 경계선을 찾아내는 알고리즘이다

이 알고리즘은 경계선에 가장 가까이 있는 support vector를 지나는 선과 거리가 최대가 되는 경계선을 구함으로써 그 목적을 달성한다.

 J는 트레이닝 데이터의 특성 값의 개수, i는 트레이닝 데이터의 개수, c는 정규화 상수, 분류 오류가 생기는 데이터에 적절한 패널티를 부여해 비선형적으로 분리되는 모델을 완화시켜 최적화된 수렴 값을 가지도록 하는 여유변수이다

이때 c의 값이 커지면 왼쪽그림처럼, c값이 작으면 오른쪽 그림처럼 경계선이 결정된다. 왼쪽은 overfitting 됐을 가능성이 크다.

from sklearn import datasets  
from sklearn.model\_selection import train\_test\_split  
from sklearn.cross\_decomposition import tests  
from sklearn.preprocessing import StandardScaler  
from sklearn.linear\_model import Perceptron  
from sklearn.linear\_model import LogisticRegression  
from sklearn.svm import SVC  
from sklearn.metrics import accuracy\_score  
import numpy as np  
from matplotlib import style  
style.use('seaborn-talk')  
  
if \_\_name\_\_ =='\_\_main\_\_':  
 iris = datasets.load\_iris()  
 X = iris.data[:, [2,3]]  
 y = iris.target  
  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=0)  
  
 sc = StandardScaler()  
 sc.fit(X\_train) # X\_train의 평균과 표준편차  
 X\_train\_std = sc.transform(X\_train) # 트레이닝 데이터 표준화  
 X\_test\_std = sc.transform(X\_test) # 테스트 데이터 표준화  
 ml = SVC(kernel='linear', C=1.0, random\_state=0)  
  
 ml.fit(X\_train\_std, y\_train)  
 y\_pred = ml.predict(X\_test\_std)  
 print('총 테스트 개수 : %d, 오류개수 : %d' %(len(y\_test), (y\_test != y\_pred).sum()))  
 print('정확도: %.2f' %accuracy\_score(y\_test, y\_pred))  
  
 X\_combined\_std = np.vstack((X\_train\_std, X\_test\_std))  
 y\_combined = np.hstack((y\_train, y\_test))

* **Unsupervised Learning**
  + **Unsupervised Learning: What is it?**

비지도 학습을 뜻한다. 지도 학습과는 달리 정답을 알려주지 않고 비슷한 데이터를 군집화 하여 미래를 예측하는 학습 방법이다. 라벨링이 되어있지 않은 데이터로부터 패턴이나 형태를 찾아야하기 때문에 어렵다. 실제로 지도 학습에서 적절한 특징(Feature)을 찾아내기 위한 전처리 방법으로 비지도 학습을 사용하기도 한다.

대표적으로 클러스터링이 있는데, 데이터는 label이나 category가 무엇인지 알 수 없는 경우가 많기 때문에 이러한 방법이 중요하다고 볼 수 있다.

예를 들어 강아지, 고양이, 기린, 원숭이 사진을 비지도 학습으로 분류한다고 가정하면, 각각의 동물들이 어떤 동물인지 정답을 알려주지 않았기 때문에 이 동물을 ‘무엇’이라고 정을 할 수 없지만 동물의 특징별로 분류하게 된다. 다리가 4개인 것은 강아지, 고양이, 기린이 한 분류를, 목이 긴 것은 기린으로 나뉘게 될 것이다.

* + **Latent Dirichlet Allocation(LDA)**

문서로부터 토픽을 추출하는 기법 중 하나로 잠재디리클레할당(LDA)이 있다. LDA란 주어진 문서에 대하여 각 문서에 어떤 주제들이 존재하는지에 대한 확률 모형인데, LDA는 토픽 별 단어의 분포, 문서 별 토픽의 분포를 모두 추정한다. 즉, 문서들은 토픽들의 혼합으로 구성되어 있고, 토픽들은 확률 분포에 기반하여 단어들을 생성한다고 가정한다.

LDA를 수행할 때 토픽이 몇 개 있는지 사용자가 정의(K)한다. K의 값을 잘못 선택하면 원치 않는 이상한 결과가 나올 수 있다. 이렇게 모델의 성능에 영향을 주는 사용자 직접 선택 매개변수를 하이퍼 파라미터라고 한다. LDA는 토픽의 제목을 정해주지 않지만, 단어의 분포를 통해 토픽을 판단해 볼 수 있다.

LDA는 문서로부터 토픽을 뽑아 내기 위해 가정을 두고 있다.

1. 문서에 사용할 단어 개수 N개
2. 문서에 사용할 토픽의 혼합을 확률 분포 기반으로 결정
3. 문서에 사용할 각 단어를 정함

아래 코드는 사이킷런을 이용한 LDA 실습 결과물이다.

from sklearn.feature\_extraction.text import TfidfVectorizer  
from sklearn.decomposition import LatentDirichletAllocation  
import pandas as pd  
import urllib.request  
import nltk  
from nltk.corpus import stopwords  
from nltk.stem import WordNetLemmatizer  
  
# https://www.kaggle.com/therohk/million-headlines 데이터 출처  
data = pd.read\_csv('abcnews-date-text.csv', error\_bad\_lines=False)  
  
text = data[['headline\_text']]  
text['headline\_text'] = text.apply(lambda row: nltk.word\_tokenize(row['headline\_text']), axis=1)  
#토큰화  
stop = stopwords.words('english')  
text['headline\_text'] = text['headline\_text'].apply(lambda x: [word for word in x if word not in (stop)])  
#불용어 처리  
text['headline\_text'] = text['headline\_text'].apply(lambda x: [WordNetLemmatizer().lemmatize(word, pos='v') for word in x])  
#표제어 추출 > 단수를 1인칭, 과거 > 현재형으로 바꿈  
tokenized\_doc = text['headline\_text'].apply(lambda x: [word for word in x if len(word) > 3])  
#길이가 3이하인 단어 제거  
detokenized\_doc = []  
for i in range(len(text)):  
 t = ' '.join(tokenized\_doc[i])  
 detokenized\_doc.append(t)  
  
text['headline\_text'] = detokenized\_doc  
#역토큰화 진행, TF-IDF에서 TfidfVectorizer 사용을 위해(토큰화 안되어있는 것을 사용)  
vectorizer = TfidfVectorizer(stop\_words='english',  
max\_features= 1000) # 상위 1,000개의 단어를 보존  
X = vectorizer.fit\_transform(text['headline\_text'])  
X.shape # TF-IDF 행렬의 크기 확인  
  
lda\_model=LatentDirichletAllocation(n\_components=10,learning\_method='online',random\_state=777,max\_iter=1)  
lda\_top=lda\_model.fit\_transform(X)  
  
terms = vectorizer.get\_feature\_names() # 단어 집합. 1,000개의 단어가 저장됨.  
  
def get\_topics(components, feature\_names, n=5):  
 for idx, topic in enumerate(components):  
 print("Topic %d:" % (idx+1), [(feature\_names[i], topic[i].round(2)) for i in topic.argsort()[:-n - 1:-1]])  
get\_topics(lda\_model.components\_,terms)

* 결과값

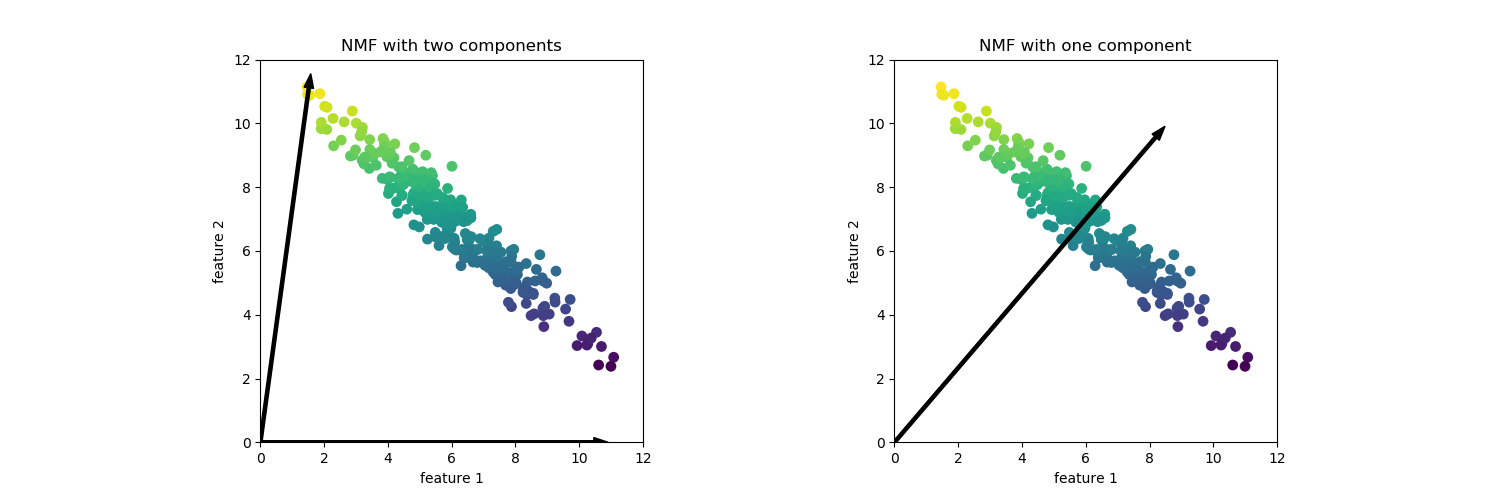
**Topic** 1: [('government', 8725.19), ('sydney', 8393.29), ('queensland', 7720.12), ('change', 5874.27), ('home', 5674.38)] **Topic** 2: [('australia', 13691.08), ('australian', 11088.95), ('melbourne', 7528.43), ('world', 6707.7), ('south', 6677.03)] **Topic** 3: [('death', 5935.06), ('interview', 5924.98), ('kill', 5851.6), ('jail', 4632.85), ('life', 4275.27)] **Topic** 4: [('house', 6113.49), ('2016', 5488.19), ('state', 4923.41), ('brisbane', 4857.21), ('tasmania', 4610.97)] **Topic** 5: [('court', 7542.74), ('attack', 6959.64), ('open', 5663.0), ('face', 5193.63), ('warn', 5115.01)] **Topic** 6: [('market', 5545.86), ('rural', 5502.89), ('plan', 4828.71), ('indigenous', 4223.4), ('power', 3968.26)] **Topic** 7: [('charge', 8428.8), ('election', 7561.63), ('adelaide', 6758.36), ('make', 5658.99), ('test', 5062.69)] **Topic** 8: [('police', 12092.44), ('crash', 5281.14), ('drug', 4290.87), ('beat', 3257.58), ('rise', 2934.92)] **Topic** 9: [('fund', 4693.03), ('labor', 4047.69), ('national', 4038.68), ('council', 4006.62), ('claim', 3604.75)] **Topic** 10: [('trump', 11966.41), ('perth', 6456.53), ('report', 5611.33), ('school', 5465.06), ('woman', 5456.76)]

* + **Non-Negative Matrix Factorization(NNMF)**

음수 미포함 행렬 분해(비음수 행렬 분해)를 뜻하며 **음수를 포함하지 않는 행렬 V를 음수를 포함하지 않은 행렬 W와 H의 곱으로 분해하는 알고리즘**이다. 행렬이 음수를 포함하지 않는 성질은 분해 결과 행렬을 찾기 쉽게 만들어 준다. 일반적으로 행렬 분해는 정확한 해가 없기 때문에 이 알고리즘은 대략적인 해를 구하게 된다. 음수 미포함 행렬 분해는 컴퓨터 시각 처리, 문서 분류, 음파 분석 등에 쓰인다.

NMF에서는 음수가 아닌 성분과 계수 값을 찾는다. 즉, 주성분과 계수가 모두 0보다 크거나 같아야 한다. 음수가 아닌 가중치 합으로 데이터를 분해하는 기능은 음수가 아닌 특성을 가진 데이터에만 적용된다.

NMF로 데이터를 다루려면 주어진 데이터가 양수인지 확인해야한다. 즉, 원점에서 상대적으로 어디에 놓여있는지가 중요하다. 그렇기에 원점에서 데이터로 가능 방향을 추출한 것으로 음수 미포함 성분을 이해할 수 있다.

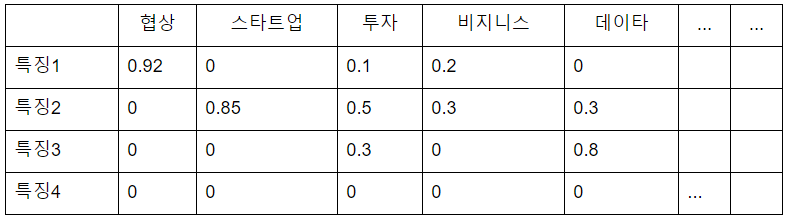
import mglearn  
import matplotlib  
import matplotlib.pyplot as plt  
  
plt.rcParams['axes.unicode\_minus'] = False # 축- 설정  
  
matplotlib.rc('font', family='AppleGothic')   
mglearn.plots.plot\_nmf\_illustration()  
  
plt.show() # 그래프 출력

데이터를 완벽하게 재구성할 수 있을 만큼 성분이 많다면(특성의 수가 많으면) 알고리즘은 데이터의 각 특성의 끝에 위치한 포인트를 가리키는 방향을 선택, 하나의 성분만 사용한다면 NMF는 데이터를 가장 잘 표현할 수 있는 평균으로 향하는 성분을 만든다.

예를 들어, 책 제목과 그 책에 나온 단어의 TFIDF 값으로 이루어진 행렬이

V가 다음과 같다면

NMF를 통해 W는 다음과 같고

H는 아래와 같은 값이 나오게 된다.

W는 가중치 행렬, H를 특성 행렬이라고 하는데, W는 책제목과 특성(특징1,2,3,4)과의 관계를, H는 원래 특성(협업, 스타트업 등)과 새로운 특성(특징1,2,3,4)의 관계를 나타낸다. 이 데이터에 ‘스타트업 데이터’도서가 새로운 데이터로 들어오게 된다면 특징 2와 3에 높은 관련성을 보일 것이다.