# 3부 : 답을 스스로 찾는 비지도 학습 알고리즘

#### 학습 목표

3부에서는 비지도 학습 알고리즘 두 개를 다룹니다. 비지도 학습은 답이 주어져 있지 않다 보니, 학습 결과가 좋은지 나쁜지 평가할 만한 답안 또한 가지고 있지 않아서 목적이 모호할 수 있습니다. 그래서 다양한 시도를 할 때 활용될 수 있습니다. 지도 학습과 달리 비지도 학습에서 압도적으로 많이 사용되는 알고리즘이 한정적입니다. 그래서 가장 유명한 두 알고리즘만 다룹니다.

# 12장 K-평균 군집화 : 카드 고객 데이터셋

#### 학습 목표

지금까지는 지도 학습에 속하는 알고리즘을 배웠습니다. 이번 장에서는 비지도 학습의 대표적인 알고리즘인 K-평균 군집화K-means Clustering를 학습합니다.

#### 학습 순서



#### K-평균 군집화 소개

K-평균 군집화는 비지도 학습의 대표적인 알고리즘 중으로 목표 변수가 없는 상태에서 데이터를 비슷한 유형끼리 묶어내는 머신러닝 기법입니다.K-최근접 이웃 알고리즘과 비슷하게 거리 기반으로 작동하며 적절한 K값을 사용자가 지정해야 합니다. 거리 기반으로 작동하기 때문에 데이터 위치가 가까운 데이터끼리 한 그룹으로 묶습니다. 이때 전체 그룹의 수는 사용자가 지정한 K개입니다.



#### 장단점

| **장점** | **단점** |
| --- | --- |
| 구현이 비교적 간단합니다. | 최적의 K값을 자동으로 찾지 못하고, 사용자가 직접 선택해야 합니다. 거리 기반 알고리즘이기 때문에, 변수의 스케일에 따라 다른 결과를 나타낼 수 있습니다. |
| 클러스터링 결과를 쉽게 해석할 수 있습니다. |  |

#### 유용한 곳

* 종속 변수가 없는 데이터셋에서 데이터 특성을 비교적 간단하게 살펴보는 용도로 활용할 수 있습니다.
* 마케팅이나 제품 기획 등을 목적으로 한 고객 분류에 사용할 수 있습니다.
* 지도 학습에서 종속 변수를 제외하고 사용하면, 탐색적 자료 분석 혹은 피처 엔지니어링 용도로 사용할 수 있습니다.

#### TOP 10 선정 이유

* 수많은 데이터를 가지고 있을 때, 데이터를 하나하나 직접 살펴보기에는 시간적인 한계가 따릅니다. 그렇다고 단순하게 통계적 정보만 살펴보는 것은 데이터를 너무 단순화하는 경향이 있습니다. 클러스터링은 이러한 상황에서 데이터를 적절한 수의 그룹으로 나누고 그 특징을 살펴볼 수 있는 장점을 제공합니다. 여러 클러스터링 기법 중에서도 K-평균 군집화는 가장 보편적이고 무난하게 사용됩니다.

## 12.1 문제 정의 : 한눈에 보는 분석 목표

<금토끼의 문제 정의> 본부장이 이번 달 매출 목표를 꼭 달성하라고 성화입니다. 금토끼도 같은 마음인 이유는 상반기 성과금이 달려있기 때문입니다. 머리를 식힐겸 옥상에 올라 커피를 홀짝 거리다 문득 아이디가 떠올랐습니다. ‘고객을 소비 형태에 따라 그룹을 나누고, 각각 알맞은 상품을 추천하면 어떨까?’ 금토끼는 곧바로 자리로 달려가 고객 1000명의 구매 이력을 분석하는 작업을 시작했습니다.

| **난이도** | ⭐☆☆ | | |
| --- | --- | --- | --- |
| **알고리즘** | K-평균 군집화(K-Means Clustering) | | |
| **데이터셋 파일명** | example\_clustering.csv  customer.csv | | |
| **데이터셋 소개** | 여기에서는 2개의 데이터를 사용합니다. 첫 번째 데이터는K-평균 군집화를 학습할 목적으로 인위적으로 만든 데이터로, 변수들에는 아무런 의미가 없습니다. 두번째 데이터는 11장에서 사용한 데이터 중 일부 변수와 일부 고객 정보만을 포함합니다. | | |
| **미션** | 데이터들을 비슷한 속성끼리 분류 | | |
| **문제 유형** | 비지도 학습 | **평가지표** | 엘보우 기법,  실루엣 점수 |
| **사용한 모델** | KMeans | | |
| **사용 라이브러리** | * numpy (numpy==1.19.5) * pandas (pandas==1.3.2) * seaborn (seaborn==0.11.2) * matplotlib (matplotlib==3.4.3) * sklearn (scikit-learn==0.23.2) * datetime, calendar | | |
| **예제 코드 노트북** | 위치 : <https://github.com/musthave-ML10/notebooks/>  파일 : 12\_KMeans.ipynb | | |

## 12.2 K-평균 군집화 맛보기 : 인위적으로 만든 데이터셋

이번에 사용할 데이터는 K-평균 군집화를 학습할 목적으로 인위적으로 만든 데이터로, 변수들에는 아무런 의미가 없습니다.

### 12.2.1 라이브러리 및 연습용 데이터 불러오기, 데이터 확인하기

연습용 데이터인 example\_cluster.csv 파일과 함께 기본적인 라이브러리들을 불러옵니다.

| import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns  file\_url = 'https://raw.githubusercontent.com/musthave-ML10/data\_source/main/example\_cluster.csv' data = pd.read\_csv(file\_url) # 데이터셋 읽기 |
| --- |

이 데이터는 K-평균 군집화를 학습할 목적으로 인위적으로 만든 데이터로, 변수들에는 아무런 의미가 없습니다. 우선 data를 출력해 데이터의 전체적인 모습을 확인해봅시다.

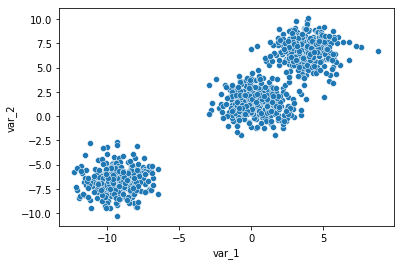
| data # 데이터 확인 |
| --- |

|  | var\_1 | var\_2 |
| --- | --- | --- |
| 0 | 3.264413 | 6.929164 |
| 1 | 0.220814 | 2.251631 |
| 2 | -8.786197 | -8.333582 |
| 3 | -0.008547 | 2.630791 |
| 4 | 4.912903 | 6.888520 |
| ... | ... | ... |
| 995 | 4.678232 | 8 |
| 996 | 5.250715 | 7 |
| 997 | -11.818752 | -6 |
| 998 | 0.613725 | 4 |
| 999 | 3.516961 | 8 |

1000 rows × 2 columns

총 데이터 1000개와 변수 2개가 있습니다. 데이터가 어떻게 분포하는지 scatterplot() 함수를 호출해 산점도 그래프를 그려 눈으로 확인하겠습니다.

| sns.scatterplot(x='var\_1', y = 'var\_2', data= data) # 산점도 그리기 |
| --- |



한눈에 보기에도 데이터가 크게 3가지 그룹으로 나뉘어 있습니다. 첫 번째 모델링에서는 이 데이터를 사람의 눈으로 인지하는 것처럼 K-평균 군집화를 이용해 3개 그룹으로 나누는 겁니다.

### 12.2.2 연습용 데이터 모델링 및 평가

K-평균 군집화는 사이킷런의 cluster에서 불러올 수 있습니다.

| from sklearn.cluster import KMeans # 임포트 |
| --- |

소개에서 언급한 바와 같이 K-평균 군집화는 만들고자 하는 그룹 수를 지정해야 합니다. n\_clusters로 지정할 수 있으며, 여기에서는 3으로 지정하여 kmeans\_model이라는 이름에 속성을 부여하겠습니다.

| kmeans\_model = KMeans(n\_clusters=3, random\_state = 100) # 모델 객체 생성 |
| --- |

이제 kmeans\_model로 학습을 시킵니다.

| kmeans\_model.fit(data) # 학습 |
| --- |

KMeans(n\_clusters=3, random\_state=100)

학습된 모델로 predict() 함수를 사용하면 데이터들을 각 클러스터로 분류합니다.,

| kmeans\_model.predict(data) # 예측 |
| --- |

array([1, 2, 0, 2, 1, 2, 1, 0, 1, 1, 1, 1, 2, 1, 0, 2, 0, 1, 1, 0, 2, 0,

0, 0, 2, 1, 2, 0, 2, 2, 2, 0, 2, 1, 1, 0, 0, 1, 1, 1, 2, 2, 1, 2,

0, 1, 1, 2, 0, 2, 1, 0, 1, 2, 1, 0, 0, 0, 2, 2, 2, 1, 2, 1, 0, 1,

2, 0, 1, 0, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 0, 2, 1, 1, 2, 1,

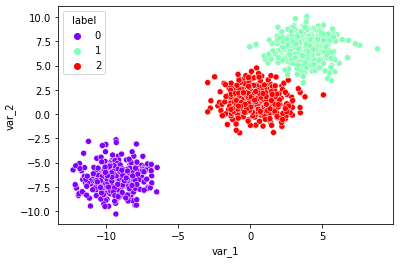
0, 2, 1, 2, 0, 2, 0, 0, 0, 1, 2, 2, 2, 0, 0, 2, 0, 1, 0, 0, 0, 1, ... 생략 ...

특정한 레이블로 구분되어 있습니다. 여기에서 0, 1, 2는 의미하는 바는 전혀 없고, 이름(레이블) 역할입니다. 즉 a, b, c 같은 역할로 이해하면 됩니다. 총 1000개의 데이터에 대한 레이블이 출력되는데, 이를 기존 데이터인 data에 label이라는 컬럼으로 붙이겠습니다.

| data['label'] = kmeans\_model.predict(data) # 예측값을 label로 저장 |
| --- |

그리고 데이터가 어떻게 나누어졌는지를 scatterplot() 함수로 확인해봅시다.

| sns.scatterplot(x='var\_1', y = 'var\_2', data= data, hue='label', palette='rainbow') # 산점도 그리기 |
| --- |



하이퍼파라미터에 hue를 사용해 레이블별로 다른 색상을 부여했습니다. palette에 rainbow를 지정해 명확히 색상으로 구분되도록 출력했습니다.

눈으로 인지했던 것과 같이 세 그룹으로 잘 나뉘어져 있습니다.

### 12.2.3 엘보우 기법으로 최적의 K값 구하기

이 데이터는 한눈에 보기에도 데이터가 3개 무리로 구성되어 있어서 K를 손쉽게 3으로 지정할 수 있지만, 실제 상황에서는 변수가 너무 많아서 그래프로 확인하기도 어려울 뿐더러, 사람의 눈으로 보기에 정확하게 몇 개의 그룹인지 구분이 애매합니다. 즉, 사람의 눈에 의존해 적절한 K값을 찾는 데 한계가 있습니다. 이러한 문제를 해결하는 방법으로 엘보우 기법elbow method을 활용할 수 있습니다.

<용어/>

**엘보우 기법**

최적의 클러스터 개수를 확인하는 방법으로, 클러스터의 중점과 각 데이터 간의 거리를 기반으로 계산합니다.

</>

각 그룹에서의 중심과 각 그룹에 해당하는 데이터 간의 거리에 대한 합을 계산합니다. 이 값을 이너셔 혹은 관성이라고 합니다. 모델이 학습을 할 때 자동적으로 이너셔값을 계산해내며, inertia\_() 함수로 확인할 수 있습니다.

<용어/>

**이너셔(inertia)**

각 그룹에서의 중심과 각 그룹에 해당하는 데이터 간의 거리에 대한 합

</>

| kmeans\_model.inertia\_ # 이너셔 확인 |
| --- |

3090.03323707666

이너셔값은 클러스터의 중점과 데이터 간의 거리이기 때문에, 작을수록 그룹별로 더 오밀조밀 잘 모이게 분류됐다고 할 수 있습니다. 즉, 작을수록 좋다고 할 수 있으나 문제는 K값이 커지면 거리의 합인 이너셔는 필연적으로 작아지게 됩니다. 예를 들어 앞에서 다룬 데이터(총 1000개)를 500개의 그룹으로 나눈다면 500개 클러스터가 각 중심을 가지고 그에 속하는 몇 개 안 되는 데이터 간의 거리를 더하기 때문에, K가 3일 때보다 더 작을 수밖에 없습니다. 직접 코드를 통해 얼마나 작아지는지 확인하겠습니다.

| temp\_model = KMeans(n\_clusters=500, random\_state = 100) # 모델 객체 생성 temp\_model.fit(data) # 학습 temp\_model.inertia\_ # 이너셔 확인 |
| --- |

6.22354210658421

K가 3일 때는 3090이던 값이, K가 500이 되니 6으로 작아졌습니다. 하지만 거리의 합이 작아졌으니 더욱 좋다고 볼수만은 없습니다. 우리가 클러스터링을 하는 이유는 각 데이터를 하나하나 살펴보기 어렵기 때문에, 거기에서 유의미한 클러스터를 만들어서 어떠한 집단적 특징을 보기 위함입니다. 그런데 만약 클러스터가 500개가 생겨버린다면 클러스터링 없이 데이터를 하나하나 확인하는 것과 별반 다르지 않은 상황이 되어버립니다. 따라서 클러스터링에서는 클러스터 수를 가급적 적게 유지하면서, 동시에 거리의 합이 어느 정도 작은, 즉 적절한 K값이 필요합니다. 이러한 이유로 K-평균 군집화에서 K값을 지정이 까다로운 겁니다.

엘보우 기법은 이에 대한 솔루션 중 하나로, 다양한 K를 넣어 모델링해보고 각각에 대한 이너셔를 구한 뒤, 이를 그림으로 풀어 적절한 K값을 찾아냅니다. K의 범위를 2부터 10까지로 지정해 모델링을 반복해봅시다.

| distance = [] # ❶ 빈 리스트 생성 for k in range(2,10): # 순회  k\_model = KMeans(n\_clusters=k) # 모델 객체 생성  k\_model.fit(data) # 학습  distance.append(k\_model.inertia\_) # ❷ 이너셔를 리스트에 저장 |
| --- |

❶ 각 이너셔값을 저장할 빈 리스트(distance)를 for문 앞에 만들었습니다. ❷ append() 함수로 각 이너셔값을 리스트(distance)에 저장합니다.

distance를 확인하면 K가 2일 때부터 10일 때까지의 각 이너셔값이 들어있습니다.

| distance # 저장된 이너셔 확인 |
| --- |

[9908.551424778394,

3090.03323707666,

2717.3943078439797,

2379.6404996021865,

2049.429711146375,

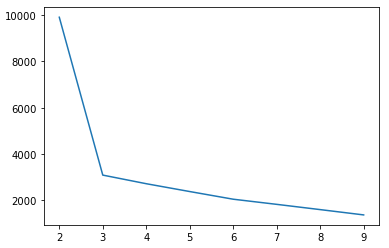
1827.5805703209694,

1596.0800725762183,

1370.1413654610615]

이제 이 값을 선형그래프로 그려보겠습니다.

| sns.lineplot(x=range(2,10), y=distance) # 엘보우 플랏 그리기 |
| --- |



x축에는 K값에 해당하는 2~10까지를, y축에는 distance를 지정했습니다.

설명했던 것처럼 K값인 x가 커짐에 따라 distance인 y는 점점 작아지는 모양입니다. 그런데 유독 급격히 각도가 변화하는 지점이 있습니다. K가 3인 지점에서 y가 크게 감소하고, 그 이후로는 조금씩 감소하여, 위의 그림이 마치 사람의 팔을 100도 정도로 접은 모습입니다. K가 3인 지점은 팔꿈치를 연상하게 하기 때문에 이를 엘보우 기법라고 부릅니다. 엘보우 기법에서는 이와 같이 distance가 급격히 줄어드는 K값을 포착하여 최적의 K값을 찾도록 도와주는 방법론입니다.

## 12.3 데이터 불러오기 및 데이터 확인하기 : 고객 데이터셋

이번에는 고객 데이터를 불러보겠습니다. 이 데이터는 11장에서 사용한 데이터 중 일부 변수와 일부 고객 정보만을 포함합니다. 파일명은 customer.csv입니다.

| file\_url = 'https://raw.githubusercontent.com/musthave-ML10/data\_source/main/customer.csv' customer= pd.read\_csv(file\_url) |
| --- |

head()를 통해 데이터가 어떻게 생겼는지 확인해봅시다.

| customer.head() # 상위 5행 출력 |
| --- |

|  | cc\_num | category | amt |
| --- | --- | --- | --- |
| 0 | 2703186189652095 | misc | 4.97 |
| 1 | 630423337322 | grocery | 107.23 |
| 2 | 38859492057661 | entertainment | 220.11 |
| 3 | 3534093764340240 | gas\_transport | 45.00 |
| 4 | 375534208663984 | misc | 41.96 |

이 데이터에는 cc\_num(카드번호)과 category(카테고리), amt(거래 금액) 변수만 있습니다. 카드번호인 cc\_num은 고객에 대한 id처럼 사용할 수 있으므로 고객이 총 몇 명이나 있는지 nunique()로 확인해봅시다.

| customer['cc\_num'].nunique() # 고윳값 확인 |
| --- |

100

고객 100명에 대한 정보가 있습니다.

이번에는 카테고리가 몇 개나 있나 살펴보겠습니다.

| customer['category'].nunique() # 고윳값 확인 |
| --- |

11

이제 각 고객이 어떤 카테고리에 얼마만큼 지불했는지를 계산한 후, K-평균 군집화로 비슷한 특성을 가진 고객끼리 묶어보겠습니다.

## 12.4 전처리 : 피처 엔지니어링

카테고리별 금액을 계산하려면 category 변수를 더미 변수로 변환시켜주어야 합니다(모든 카테고리를 분석 결과에서도 확인할 수 있도록 더미 변수 하나를 줄이는 drop\_first 매개변수는 사용하지 않겠습니다). 카테고리가 11개이므로 더미 변수 11개가 추가될 겁니다.

| customer\_dummy = pd.get\_dummies(customer, columns =['category']) # 더미 변수로 변환 |
| --- |

head() 함수를 호출해 확인해봅시다.

| customer\_dummy.head() # 상위 5행 확인 |
| --- |

|  | cc\_num | amt | category\_entertainment | category\_food\_dining | category\_gas\_transport | category\_grocery | category\_health\_fitness | category\_home | category\_kids\_pets | category\_misc | category\_personal\_care | category\_shopping | category\_travel |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 2703186189652095 | 4.97 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 630423337322 | 107.23 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 38859492057661 | 220.11 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 3534093764340240 | 45 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 375534208663984 | 41.96 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

카테고리별로 얼마만큼의 금액을 썼는지를 계산하기 위해, 더미 변수로 변환된 영역에 amt(거래 금액) 변수의 값을 곱하여 계산합니다. 더미 변수들은 특정 카테고리에 해당하면 1, 해당 사항이 없으면 0이 되므로 거래 금액인 amt 변수를 곱하면 자연스럽게 사용된 변수에 금액이 들어갑니다.

우선 더미로 변환된 변수들의 이름을 하나의 리스트로 모아야 합니다. 컬럼 이름을 불러오는 columns를 이용하여 cc\_num과 amt를 제외하도록 다음과 같이 인덱싱합니다. 이를 cat\_list라는 이름으로 저장하겠습니다.

| cat\_list = customer\_dummy.columns[2:] # 변수 이름 리스트 생성 |
| --- |

이 리스트에 있는 변수 이름들을 for문으로 하나씩 불러와서 sum 변수와 곱한 후, 그 결과를 기존 이름에 그대로 덮어쓰겠습니다.

| for i in cat\_list:  customer\_dummy[i] = customer\_dummy[i] \* customer\_dummy['amt'] # 금액으로 변수 업데이트 |
| --- |

이제 customer\_dummy을 확인하면 각 더미 변수 자리에 1이 아닌 금액이 들어있습니다.

| customer\_dummy |
| --- |

|  | cc\_num | amt | category\_entertainment | category\_food\_dining | category\_gas\_transport | category\_grocery | category\_health\_fitness | category\_home | category\_kids\_pets | category\_misc | category\_personal\_care | category\_shopping | category\_travel |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 2703186189652095 | 4.97 | 0.00 | 0.00 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 4.97 | 0.0 | 0.0 | 0.0 |
| 1 | 630423337322 | 107.23 | 0.00 | 0.00 | 0.0 | 107.23 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 38859492057661 | 220.11 | 220.11 | 0.00 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3 | 3534093764340240 | 45.00 | 0.00 | 0.00 | 45.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 4 | 375534208663984 | 41.96 | 0.00 | 0.00 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 41.96 | 0.0 | 0.0 | 0.0 |

이제 각 거래 건으로 정리된 데이터를 고객 레벨로 취합하는 데 groupby() 함수를 사용하겠습니다. groupby()에 cc\_num을 넣어주고 sum()으로 계산하면 고객별 총 사용 금액 및 카테고리별 사용 금액을 구할 수 있습니다.

| customer\_agg = customer\_dummy.groupby('cc\_num').sum() # cc\_num별 총 금액 |
| --- |

계산된 테이블을 head()로 확인해봅시다.

| customer\_agg.head() # 상위 5행 확인 |
| --- |

| cc\_num | amt | category\_entertainment | category\_food\_dining | category\_gas\_transport | category\_grocery | category\_health\_fitness | category\_home | category\_kids\_pets | category\_misc | category\_personal\_care | category\_shopping | category\_travel |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 60495593109 | 48,149.62 | 3,169.74 | 4,043.46 | 5,076.36 | 7,576.58 | 1,631.22 | 5,458.58 | 4,924.97 | 4,889.57 | 3,135.25 | 8,106.36 | 137.53 |
| 571365235126 | 259,784.87 | 13,864.51 | 9,055.51 | 36,525.79 | 48,812.73 | 13,685.60 | 19,342.79 | 17,726.82 | 32,542.61 | 16,672.08 | 45,994.60 | 5,561.83 |
| 571465035400 | 270,081.63 | 16,162.98 | 15,448.75 | 9,081.62 | 41,643.82 | 15,244.03 | 28,282.98 | 13,737.95 | 15,880.82 | 8,771.27 | 59,454.96 | 46,372.45 |
| 630412733309 | 41,959.52 | 3,212.46 | 1,543.16 | 5,786.98 | 8,416.70 | 3,201.04 | 2,474.68 | 3,648.97 | 3,109.83 | 2,592.17 | 7,586.79 | 386.74 |
| 630423337322 | 244,612.73 | 12,786.75 | 6,905.07 | 36,460.40 | 43,854.26 | 12,136.74 | 21,106.87 | 16,807.24 | 32,400.76 | 15,885.32 | 44,287.02 | 1,982.30 |

원하는 형태로 데이터가 잘 정리되었습니다. 이 다음으로는 스케일링을 해줘야 합니다. K-평균 군집화는 거리 기반 알고리즘이기 때문에 데이터의 스케일에 영향을 받습니다. 여기서는 아웃라이어가 유의미한 역할을 할 수 있습니다. 예를 들어 특정 고객이 특정 카테고리에 너무 많은 돈을 사용했다면 아웃라이어로 분류하여 별도의 처리를 하기보다는, 그 특성 그대로를 남겨두어서 클러스터링에 반영되게 하는 것이 좋습니다. 그러므로 아웃라이어 효과를 감소시키지 않는 StandardScaler를 사용하겠습니다.

| from sklearn.preprocessing import StandardScaler # 임포트 scaler = StandardScaler() # ❶ 스케일러 객체 생성 scaled\_df = pd.DataFrame(scaler.fit\_transform(customer\_agg), # ❷  columns = customer\_agg.columns, # ❸  index=customer\_agg.index) # ❹ 스케일링 후 데이터프레임으로 변환 |
| --- |

❶ 스케일러의 속성을 scaler라는 이름에 부여합니다.

❷ scaler의 fit\_transform()으로 customer\_agg를 변환시키며, 이를 데이터 프레임 형태로 만듭니다

❸ 데이터 프레임의 컬럼 이름을 정의합니다.

❹ 데이터 프레임의 인덱스 이름을 정의합니다.

마지막으로 scaled\_df를 head()를 호출해 확인하겠습니다.

| scaled\_df.head() # 상위 5행 출력 |
| --- |

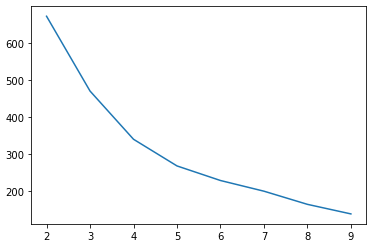
| cc\_num | amt | category\_entertainment | category\_food\_dining | category\_gas\_transport | category\_grocery | category\_health\_fitness | category\_home | category\_kids\_pets | category\_misc | category\_personal\_care | category\_shopping | category\_travel |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 60495593109 | -1.40 | -1.14 | -0.97 | -1.00 | -1.12 | -1.56 | -1.15 | -1.28 | -1.12 | -1.07 | -1.14 | -0.62 |
| 571365235126 | 1.08 | 0.41 | 0.17 | 2.09 | 0.77 | 1.04 | 0.84 | 1.13 | 1.21 | 1.81 | 0.66 | -0.22 |
| 571465035400 | 1.20 | 0.75 | 1.62 | -0.61 | 0.44 | 1.38 | 2.13 | 0.38 | -0.20 | 0.13 | 1.30 | 2.77 |
| 630412733309 | -1.47 | -1.13 | -1.54 | -0.93 | -1.08 | -1.22 | -1.57 | -1.52 | -1.27 | -1.19 | -1.17 | -0.60 |
| 630423337322 | 0.90 | 0.26 | -0.32 | 2.08 | 0.54 | 0.71 | 1.10 | 0.96 | 1.20 | 1.64 | 0.58 | -0.48 |

다른 고객들과 비슷한 수준이면, 즉 평균에 가까울 경우는 0에 근접한 값을 보여주게 되고, 더 많이 사용했으면 더 큰 양수를, 더 적게 사용했으면 더 작은 음수를 갖습니다. 이제 이 데이터에 K-평균 군집화 알고리즘을 적용하겠습니다.

## 12.5 고객 데이터 모델링 및 실루엣 계수

이번에는 적절한 K값을 전혀 예상할 수 없기 때문에 바로 엘보우 기법을 사용하겠습니다. K의 범위는 마찬가지로 2~10으로 설정하겠습니다.

| distance = [] # 빈 리스트 생성 for k in range(2,10): # 순회  k\_model = KMeans(n\_clusters=k) # 모델 객체 생성  k\_model.fit(scaled\_df) # 학습  labels = k\_model.predict(scaled\_df) # 예측  distance.append(k\_model.inertia\_) # 이너셔값 리스트에 추가   sns.lineplot(x=range(2,10), y=distance) # 엘보우 플랏 그리기 |
| --- |



어느 한 지점에서 크게 떨어지지 않고 비교적 완만하게 그래프가 내려가고 있습니다. K값을 결정하기가 상당히 어려운 모양새입니다. 이는 실제 프로젝트에서 굉장히 빈번하게 나타나는 현상입니다. 이에 대한 대안으로 실루엣 계수silhouette coefficient를 사용할 수 있습니다. 엘보우 기법에서 사용되는 이너셔는 각 클러스터의 중심과 그 클러스터에 속한 데이터 간의 거리로만 계산되는 반면, 실루엣 계수는 클러스터 내부에서의 평균 거리와, 최근접한 다른 클러스터 데이터와의 평균 거리도 점수에 반영합니다. 그림으로 설명하면 다음과 같습니다.

<용어/>

**실루엣 계수**

엘보우 기법과 같이 최적의 클러스터 수를 찾는 방법으로, 엘보우 기법에서 적절한 클러스터 수를 찾지 못했을 때 대안으로 사용할 수 있습니다. 엘보우 기법보다 계산 시간이 오래 걸리는 단점이 있습니다.

</>

<그림/>



클러스터 A

클러스터 B

클러스터 C

</>

클러스터 A를 중심으로 계산하겠습니다. 일단 내부 데이터와 중심과의 평균을 다음과 같이 a로 정의합니다.

그리고 클러스터 A의 중심과 클러스터 B, 클러스터 C에 속한 데이터의 거리에 대한 평균을 구합니다. 이중 값이 더 작은 것, 즉 가까운 클러스터와의 거리를 b로 정의합니다. 클러스터 B와의 거리가 더 가까우므로 b는 13이 됩니다.

a, b를 구했으면 실루엣 계수는 아래 수식으로 계산합니다.

이와 같은 방식으로 각 클러스터에 대해 계산하여 총 합산한 값이 최종 실루엣 계수가 됩니다. 이를 코딩으로 구현하기 위해 사이킷런의 metrics에서 silhouette\_score를 불러옵니다.

| from sklearn.metrics import silhouette\_score # 임포트 |
| --- |

silhouette\_score() 함수에 K-평균 군집화 학습에 사용한 원본 데이터와 그 예측값(label)을 차례로 넣어주면 됩니다. 즉, 다음과 같은 코드로 표현할 수 있습니다.

| silhouette = [] # 빈 리스트 생성 for k in range(2,10): # 순회  k\_model = KMeans(n\_clusters=k) # 모델 객체 생성  k\_model.fit(scaled\_df) # 학습  labels = k\_model.predict(scaled\_df) # 예측  silhouette.append(silhouette\_score(scaled\_df, labels)) 실루엣 계수 리스트에 추가 |
| --- |

엘보우 기법과 마찬가지로 for문을 활용하여 다양한 K값을 넣어보고 이에 따른 각 실루엣 계수를 리스트(silhouette)에 저장했습니다.

이제 silhouette 값과 K값의 범위 2~10을 이용하여 선형 그래프를 그려보겠습니다.

| sns.lineplot(x=range(2,10), y=silhouette) # 선형 그래프 그리기 |
| --- |



실루엣 계수에서는 높은 값일수록 더 좋은 분류를 의미한다고 했습니다. 따라서 여기에서는 명확하게 K가 4일 때 가장 좋은 분류 성능을 냅니다.

실루엣 계수의 단점은 계산 비용이 상대적으로 크다는 겁니다. 여기서는 엘보우 기법이 이너셔 기법보다 다소 더 시간이 걸렸습니다. 데이터가 상당히 크다면 생각보다 오랜 시간을 기다려야 했을 겁니다. 기본적으로는 엘보우 기법을 우선 활용하고, 적절한 K값을 찾기 어려울 때 실루엣 계수를 쓰기 바랍니다.

## 12.6 최종 예측 모델 및 결과 해석

K값을 찾았으면 해당 값을 넣어 다시 한번 모델링한 뒤, 해당 모델을 사용하여 레이블을 구합시다.

| k\_model = KMeans(n\_clusters=4) # 모델 객체 생성 k\_model.fit(scaled\_df) # 학습 labels = k\_model.predict(scaled\_df) # 예측 |
| --- |

이제 labels를 기존의 데이터프레임 scaled\_df에 새로운 컬럼으로 붙여줍니다.

| scaled\_df['label'] = labels # label 변수 정의 |
| --- |

클러스터링 결과를 해석하려면 label별로 데이터를 요약할 겁니다. 전체 금액 및 카테고리 금액에 대해서 label별 평균값을 구하고, 각 label에 몇 명의 고객이 있는지도 확인해봅시다.

| scaled\_df\_mean = scaled\_df.groupby('label').mean() # ❶ label별 평균값 scaled\_df\_count = scaled\_df.groupby('label').count()['category\_travel'] # ❷ label별 등장 횟수 |
| --- |

❶ 전체 금액 및 카테고리 금액에 대한 label별 평균값을 scaled\_df\_mean에 저장합니다. ❷ groupby()를 사용해 label별 고객 수를 scaled\_df\_count에 저장합니다. groupby()와 count()의 조합은 해당 데이터프레임의 모든 변수에 대한 count() 값을 보여줍니다. 모든 변수가 동일한 행을 가지고 있으므로 같은 값을 보이고, 우리는 이중 임의의 한 변수에 대한 count() 값만 있으면 됩니다. 따라서 [‘category\_travel’]로 임의의 한 변수만 불러옵니다. 아무 변수 이름이나 써주셔도 됩니다.

scaled\_df\_mean과 scaled\_df\_count 모두에 category\_travel이라는 컬럼이 존재하기 때문에, 바로 join() 함수를 사용하여 결합하면 에러가 발생합니다. 이런 이유로 scaled\_df\_count의 컬럼명을 미리 수정하겠습니다.

| scaled\_df\_count = scaled\_df\_count.rename('count') # 이름 변경 |
| --- |

판다스 시리즈에는 컬럼이 하나뿐이므로 rename 안에 딕셔너리 타입을 쓸 필요 없이 바로 원하는 이름만 써주면 됩니다. 여기서는 ‘count’라는 이름으로 변경하겠습니다.

이제 두 데이터를 join() 함수로 결합하여 최종 데이터프레임을 완성시킵니다.

| scaled\_df\_all = scaled\_df\_mean.join(scaled\_df\_count) # 데이터 합치기 |
| --- |

완성된 데이터를 출력하면 다음과 같습니다.

| scaled\_df\_all |
| --- |

| label | amt | category\_entertainment | category\_food\_dining | category\_gas\_transport | category\_grocery | category\_health\_fitness | category\_home | category\_kids\_pets | category\_misc | category\_personal\_care | category\_shopping | category\_travel | count |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | -0.858082 | -0.739555 | -0.758078 | -0.601061 | -0.653905 | -0.837854 | -0.832432 | -0.8661 | -0.701383 | -0.760106 | -0.689347 | -0.266604 | 45 |
| 1 | 0.311574 | 0.149105 | 0.212407 | 1.025558 | 0.10907 | 0.452339 | 0.369237 | 0.716395 | 0.378376 | 0.47004 | 0.010316 | -0.26441 | 37 |
| 2 | 1.90812 | 2.155375 | 1.796299 | -0.490418 | 2.290553 | 1.155923 | 0.94395 | 1.255997 | 2.005393 | 1.75621 | 2.272314 | -0.26274 | 10 |
| 3 | 1.000533 | 0.776166 | 1.036432 | -0.749213 | 0.310577 | 1.175961 | 1.794775 | -0.01151 | -0.311453 | -0.093603 | 0.989477 | 3.05097 | 8 |

클러스터 0(label이 0인 그룹)은 전체 지출액부터 각 카테고리의 금액까지 모두 타 클러스터에 비해서 낮습니다. 여기에 속하는 고객 수는 약 45명으로 가장 많습니다. 클러스터 1은 gas\_transfport에서 상대적으로 조금 높은 지출을 보이지만, 전체 지출 및 각 카테고리에서 대부분 0에 가까운 숫자를 보였습니다. 총 37명으로 딱히 특징이 없는 일반 고객군으로 분류할 수 있습니다. 클러스터 2는 전체 지출금액부터 대다수의 카테고리에서 높은 지출을 보여줍니다. 총 10명의 고객으로 상위 10%의 VIP 고객이라 볼 수 있습니다. 마지막으로 클러스터 3은 health, home, travel에서 다른 클러스터보다 지출이 높습니다. 그중에서도 travel이 유일하게 굉장히 높게 나온 클러스터로, 여행에 관심이 많은 고객군으로 정의할 수 있습니다. 클러스터 1에 해당하는 고객은 총 8명입니다.

## 12.7 이해하기 : K-평균 군집화

K-평균 군집화는 각 클러스터의 중점과 각 데이터와의 거리를 기반으로 클러스터를 정의합니다. 임의의 데이터를 그려서 그림과 함께 설명드리겠습니다.

아래 예시에는 총 14개의 데이터가 있습니다(이터레이터 0). K=3으로 가정하여 총 3개의 클러스터를 만들겠습니다. K-평균 군집화는 중심점 K개를 임의로 설정합니다(이터레이터 1). 그리고 각 중심점을 기준으로 가까이에 있는 데이터들을 해당 클러스터로 할당합니다(이터레이터 2).

<그림>



</>

이렇게 랜덤하게 3개의 초기 클러스터가 나뉘어졌습니다. 이제 각 클러스터에 속한 데이터의 중점을 재계산하여 X의 위치를 움직입니다(이터레이터 3). 이동한 중점을 기준으로 하여 중점과 데이터 간의 거리를 재계산하고, 중점과 가까운 쪽으로 클러스터를 재정의합니다(이터레이터 4). 또 다시 이동한 중점을 기반으로 클러스터링을 재계산하고, 더 이상 클러스터에 변동이 없을 때까지 이 과정을 계속하여 반복합니다.

<그림>



</>

처음에는 임의로 배치된 클러스터로 시작됐지만, 위와 같은 과정을 반복하여 마지막에는 그럴듯하게 3개의 클러스터로 분리되었습니다.

## 학습 마무리

#### 되짚어보기

12.1 머신러닝 문제 2개를 풀어봅니다. 첫 번째는 임의로 만든 데이터셋을 활용해 K-평균 군집화의 개념을 살펴봅니다. 두 번째는 K-평균 군집화 알고리즘을 사용해 카드 사용 고객의 소비 형태에 따라 그룹을 나눕니다.

12.2 연습용 데이터를 이용해 K-평균 군집화 알고리즘을 맛보겠습니다.

임의의 데이터셋을 사용합니다. K-평균 군집화 알고리즘으로 데이터를 3개 그룹으로 분류합니다. 산점도로 결과를 확인해보니 기대한 대로 잘 분류되었습니다. 엘보우 기법을 사용하여 최적의 그룹 수를 알아봅니다.

12.3 고객 카드 사용 이력 데이터셋을 불러옵니다.

12.4 카테고리 변수를 더미 변수로 변환해 각 카테고리별 사용금액을 계산합니다. 군집화 이전에 스케일을 조정해주었습니다.

12.5 고객 데이터에서는 엘보우 기법으로 적절한 그룹 수를 찾을 수 없었습니다. 그래서 실루엣 계수를 활용하여 최적의 그룹 수를 찾았습니다.

12.6 실루엣 계수를 통해 4개의 그룹이 최적임을 확인했습니다. 고객들을 4개 그룹으로 분류하고 각각 어떤 특성이 있는지를 확인해보았습니다.



#### 과제

4장의 타이타닉 데이터셋를 가지고 클러스터링을 연습해봅시다. 어떤 특성을 가진 승객들로 묶이는지 확인할 수 있습니다. KMeans 클러스터링은 학습 지도에서 피처 엔지니어링으로 사용할 수도 있습니다. 이 장에서 구한 고객별 클러스터를 11장의 피처엔지니어링에서 하나의 변수로 추가하여 활용해봅시다.

#### 유의할 점

#### 관련 모델

1. **DBScan**  
   패키지: from sklearn.cluster import DBSCAN  
   클러스터링을 위한 밀도 기반의 알고리즘입니다.

#### 핵심 용어 정리

1. **K-평균 군집화** : 데이터를 거리 기반으로 측정하여 가까이 있는 데이터들을 하나로 묶어주는 방법입니다. 예측하려는 종속변수가 존재하지 않을 때 사용하는 비지도 학습의 대표적인 알고리즘입니다.
2. **엘보우 기법** : 최적의 클러스터 개수를 확인하는 방법으로, 클러스터의 중점과 각 데이터 간의 거리를 기반으로 계산합니다.
3. **이너셔** : 각 클러스터의 중점과 그에 속한 데이터 간의 거리. 값이 작을수록 잘 뭉쳐진 클러스터를 의미합니다.
4. **실루엣 계수** : 엘보우 기법과 같이 최적의 클러스터 수를 찾는 방법으로, 엘보우 기법에서 적절한 클러스터 수를 찾지 못했을 때 대안으로 사용할 수 있습니다. 엘보우 기법보다 계산 시간이 오래 걸리는 단점이 있습니다.

#### 새로운 함수와 라이브러리

* **sklearn.metrics.silhouette\_score()** : 클러스터링의 평가지표인 실루엣 계수를 계산합니다.
* **KMeans모델.inertia\_()** : 학습된 Kmeans 모델에서 이너셔 값을 호출합니다.

## 연습문제

1. 다음 K-평균 군집화 설명 중 옳지 않은 것은?

① 비지도 학습으로 정해진 답이 없이 수행된다.

② 최적의 K값은 알고리즘에서 스스로 찾아낼 수 있다.

③ 최초에는 임의로 배치된 클러스터로 시작됩니다.

④ 거리 기반으로 작동하는 알고리즘이다.

2. 엘보우 기법과 실루엣 계수에 대한 설명으로 옳은 것은?

① 최적의 K값을 찾아내는 방법들이다.

② 실루엣 계수가 잘 작동하지 않으면 엘보우 기법을 사용해볼 수 있다.

③ 엘보우 기법에서는 가장 낮은 값을 찾아내어 최적의 K값을 찾는다.

④ 실루엣 계수에서는 가장 높은 값을 찾아내어 최적의 K값을 찾는다.

3. 시본 산점도에서 특정 변수에 따라 색상을 달리 정의하는 매개변수는?

① size

② marker

③ palette

④ hue

#### 정답 및 해설

1. 2

② 최적의 K값은 알고리즘에서 스스로 찾아낼 수 있다. ← 최적의 K값은 알고리즘이 정하지 못하므로 엘보우 기법이나 실루엣 계수로 찾아내야 합니다.

2. 3

③ 엘보우 기법에서는 가장 낮은 값을 찾아내어 최적의 K값을 찾는다. ← 가장 낮은 값이 아니라 급격하게 꺾이는 부분을 포착해야 합니다.

3. 4

① size ← 각 점의 크기를 결정합니다.

② marker ← 각 점의 모양을 결정합니다.

③ palette ← 색상의 구성을 결정합니다.