제2차 빅데이터 분석 교육과정 강의교재

- 가완성본 -

ABRC 인공지능-빅데이터연구센터
AI-BIG DATA RESEARCH CENTER

K-Means와 KNN

1. K-means

1.1 K-means 알고리즘이란?

1.1.1 K-menas 알고리즘 소개

- 비지도 학습(Unsupervised Learning)의 한 종류인 클러스터링(Clustering)의 대표적인 알고리즘
- 주어진 데이터를 K개의 클러스터(Cluster)로 묶는 알고리즘, 데이터들과 각 클러스터와의 거리 차이의 분산을 최소화
- Python에서 대표적으로 scikit-learn(sklearn) 패키지에서 K-menas 클러스터링을 위한 함수 제공

- 비지도 학습(Unsupervisde Learning)
  - 훈련 데이터(Training Data)로부터 하나의 함수를 추론하는 방법 중 하나로, 지도학습과 달리 데이터에서 추출하고자 하는 라벨(Label)이 없는 데이터를 이용해 함수를 추론하는 것으로 일반적으로 클러스터링(군집)을 위해 사용
- 클러스터링(군집)
  - 라벨(Label) 데이터 없이 데이터 각각의 특성을 고려해 주어진 데이터를 가장 잘 설명하는 집단(클러스터)을 찾아 속하게 하는 것(나누는 것)

K-means는 비지도학습의 한 종류인 클러스터링의 대표적인 알고리즘입니다. 비지도 학습은 지도 학습과 달리 데이터에서 추출하고자 하는 라벨이 없는 데이터를 이용해 함수를 추론하는 방법입니다. 그렇다면 클러스터링이란 무엇일까요? 클러스터링은 말 그대로 여러 개의 객체를 모아 집단으로 만드는 것입니다.

즉, 머신러닝에서 클러스터링이란 라벨이 없는 데이터 각각의 특성을 고려하여 주어진 데이터들을 가장 잘 설명하는 몇 개의 집단으로 나누는 것을 의미합니다. 클러스터링의 대표적 알고리즘인 K-means는 데이터를 K개의 클러스터, 즉 집단으로 묶는 알고리즘이며 K 값에 따라 결과가 달라집니다. 집단을 클러스터라고 부르도록 하겠습니다.

그렇다면 K개의 클러스터로 나누는 기준은 무엇일까요? 바로 데이터들 간의 유사도를 판단하는데요, 이는 데이터들 간의 거리를 측정하여 계산하게 됩니다. K-means 클러스터링을 위한 함수를 제공하는 Python의 scikit-learn 패키지를 이용해 실습해 볼 것입니다. 그 전에 먼저, k-means 알고리즘에 대해 자세히 알아볼까요? 먼저 알고리즘의 원리를 살펴보겠습니다.

1.1.2 K-means 알고리즘 원리

(1) 클러스터 내 응집도 최소화

- 클러스터 1(G_1) 데이터와 클러스터 1의 중심 값(C_1)과의 거리 합 최소화
- 클러스터 2(G_2) 데이터와 클러스터 2의 중심 값(C_2)과의 거리 합 최소화

$$Min\sum_{i=1}^{K}\sum_{x\in G_i} d(C_i,x)$$

(2) 클러스터 간 분리도 최대화

- 클러스터 1(G_1)의 중심 값(C_1)과 클러스터 2(G_2)의 중심 값(C_2)과의 거리 최대화

$$Max\sum_{i,j=1}^{k} d(c_i,c_j), i \neq j$$

알고리즘은 크게 두 가지 원리를 바탕으로 동작합니다. 먼저 클러스터 내 응집도를 최소화하는 것입니다. 쉽게 설명하면 하나의 클러스터 안에 속한 데이터들 간의 거리의 합을 최소화시키는 것을 의미합니다. 여기서 중요한 개념이 나옵니다.

앞에서 데이터 간의 거리 측정을 통해 유사도를 측정한다고 설명했는데요, 여기서 거리 측정은 '센트로이드'라고 불리는 클러스터의 중심점과 각 데이터 간에 이루어집니다. 클러스터의 중심점은 말 그대로 클러스터 내의 데이터와 클러스터 중심점 간의 거리가 최소가 되는 점으로 여러 번의 학습을 통해 이 중심점을 찾게 됩니다.

두 번째로, 클러스터 간 분리도 최대화입니다. 이는 다른 클러스터 간의 중심점의 거리가 최대화되어야 한다는 것입니다. 그래야 각각의 클러스터를 구분하기 쉽겠죠? 이번에는 데이터 간의 거리 측정법에 대해 알아보겠습니다.

1.1.3 거리 측정법

- 중심에서 멀리 떨어져 있어도, K-means 알고리즘은 유클리드 거리 측정법을 사용해서 반복적으로 중심을 수정해나갈 수 있음
- 유클리드 거리 측정법을 점 간의 거리 산출에 이용
- 벡터 사이에 거리가 짧으면 유사성이 더 높다는 것을 의미

데이터 간의 유사도 측정을 위한 거리 계산법에는 여러 가지가 있는데, K-means에서는 유클리드 거리 측정법을 주로 사용합니다.

1.1.4 유클리드 거리 측정법

$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2}$$

- 유클리드 거리가 큰 값을 가지면 사용자 간의 거리가 멀다는 의미이기 때문에 사용자 간 유사성이 떨어짐을 의미함

유클리드 거리 측정법은 벡터 간의 거리가 짧으면 둘 사이의 유사성이 더 높다는 것을 전제로 합니다. 수식을 보면 각 벡터 간의 차를 제곱한 합계 루트 값으로 거리를 측정하게 됩니다. 이제 K-means 알고리즘이 어떻게 수행되는지 단계별로 설명 드리도록 하겠습니다.

1.1.5 K-means 알고리즘 수행 단계

(1) 전체 클러스터의 개수 K값을 설정한다.

(2) (초기 중심점 선택) K개의 그룹으로 군집한 n개의 점을 가지고 있고, 여기서 임의로 K개의 점을 선택하여 K개의 클러스터의 초기 중심점(Centroid)을 정한다.

(3) (클러스터 할당) K개의 클러스터의 중심점과 각 개별 점(데이터) 간의 거리를 측정하여 가장 가까운 클러스터에 해당 데이터를 할당한다.

- 여기서 거리 측정은 '유클리드 거리 측정법'을 사용한다.
- 이 때, 데이터는 다른 클러스터 중심 값들보다 할당된 클러스터의 중심 값과 가장 가깝다.

(4) (새로운 중심점 선택) 각 클러스터 마다 그 안에 배정된 모든 점들 간의 거리 평균값을 구하여 새로운 중심점으로 정한다.

(5) 만약, 새로운 중심 값들이 이전의 중심 값들과 동일하다면 알고리즘을 끝내고 그렇지 않으면 (3) ~ (5) 단계를 계속 반복한다.

먼저 K-means 알고리즘은 사용자가 K값, 즉 클러스터의 개수를 임의로 정합니다. 두 번째로, K개의 점을 임의로 선택하여 K개 클러스터의 초기 중심 값으로 정합니다. 그리고 K개의 클러스터 중심점과 각 데이터 간의 거리를 측정하고 각 데이터는 자신과 가장 가까운 클러스터 중심점의 클러스터에 할당됩니다.

이제 클러스터 내 응집도를 최소화하기 위해 각 클러스터의 중심점과 여기에 할당된 모든 데이터들 간의 거리의 평균값을 가지는 점을 선택하여 새로운 중심점으로 변경합니다. (3), (4), (5)번은 K개의 중심점이 변하지 않을 때까지 반복하게 되고, 어느 정도의 오차 범위 안에 들어오게 되면 반복을 중단하게 됩니다. 이제 하나의 예제를 가지고 좀 더 자세히 설명해보겠습니다.

1.2 K-means 클러스터링 예제

1.2.1 예제 1 : K-means를 이용한 단순 클러스터링 예제

(1) 초기 데이터 및 클러스터 개수 K 설정

데이터	x축	y축
1	1	1
2	2	1
3	1	2
4	2	2
5	3	3
6	8	8
7	8	9
8	9	8
9	9	9

- 클러스터 개수 K : 2
- 클러스터 대상 데이터 : (1, 1), (2, 1), (1, 2), (2, 2), (3, 3), (8, 8), (8, 9), (9, 8), (9, 9) 총 9개
- 거리 측정법 : 유클리드 거리 측정법

주어진 데이터는 총 9개로 x, y 두 개의 속성 값을 숫자 형태로 가지고 있습니다. 이 9개의 데이터를 임의로 설정한 2개의 클러스터로 나누겠습니다.

1.2.2 초기 클러스터 중심점 선택

클러스터 중심점	데이터
(1, 1) (2, 1)	1, 3 2, 4, 5, 6, 7, 8, 9
(1, 1.5) (5.85, 5.71)	1, 2, 3, 4, 5 6, 7, 8, 9
(1.8, 1.8) (8.5, 8.5)	1, 2, 3, 4, 5 6, 7, 8, 9

먼저 전체 데이터를 임의로 2개의 클러스터로 나눈 후 중심점을 선택합니다. 우리는 (1, 1), (2, 1) 두 개를 각 클러스터의 중심점으로 선택하겠습니다. 이해를 돕기 위해 전체 데이터를 2차원 평면에 나타내 보았습니다. 데이터의 각 x, y값은 x축, y축 값으로, 데이터는 점으로 표현하였습니다. 그림 1은 클러스터의 초기 중심점과 데이터를 그래프로 표현했습니다.

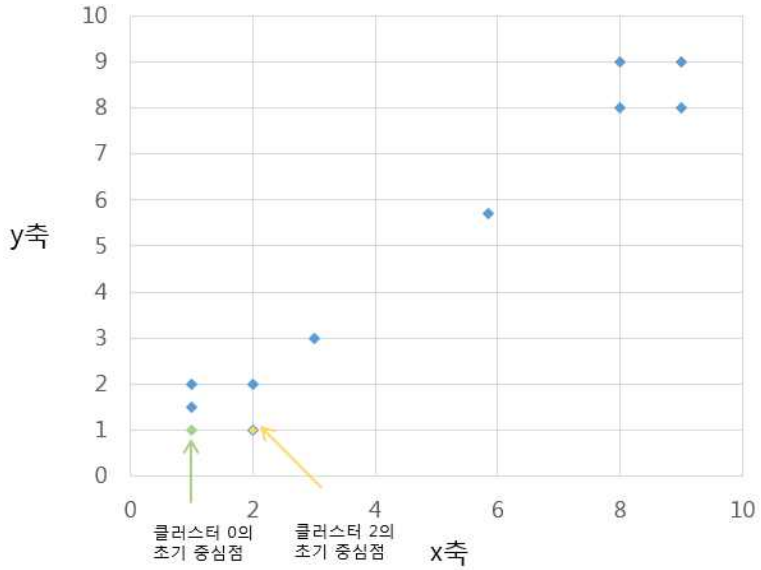


그림 1 클러스터의 초기 중심점과 데이터 그래프 표현

(3) 각 클러스터의 중심점과의 거리 계산

데이터	클러스터 0의 중심점 (1, 1)과의 거리	대소 비교	클러스터 1의 중심점 (2, 1)과의 거리
1	$d=\sqrt{(1-1)^2+(1-1)^2}=0$	<	$d=\sqrt{(2-1)^2+(1-1)^2}=1$
2	$d=\sqrt{(1-2)^2+(1-1)^2}=1$	>	$d=\sqrt{(2-2)^2+(1-1)^2}=0$
3	$d=\sqrt{(1-1)^2+(1-2)^2}=1$	<	$d=\sqrt{(2-1)^2+(1-2)^2}=\sqrt{2}=1.414$
4	$d=\sqrt{(1-2)^2+(1-2)^2}=\sqrt{2}=1.414$	>	$d=\sqrt{(2-2)^2+(1-2)^2}=1$
5	$d=\sqrt{(1-3)^2+(1-3)^2}=\sqrt{8}=2.82$	>	$d=\sqrt{(2-3)^2+(1-3)^2}=\sqrt{5}=2.23$
6	$d=\sqrt{(1-8)^2+(1-8)^2}=\sqrt{98}=9.89$	>	$d=\sqrt{(2-8)^2+(1-8)^2}=\sqrt{85}=9.21$
7	$d=\sqrt{(1-8)^2+(1-9)^2}=\sqrt{113}=10.63$	>	$d=\sqrt{(2-8)^2+(1-8)^2}=\sqrt{100}=10$
8	$d=\sqrt{(1-9)^2+(1-8)^2}=\sqrt{113}=10.63$	>	$d=\sqrt{(2-9)^2+(1-8)^2}=\sqrt{98}=9.89$
9	$d=\sqrt{(1-9)^2+(1-9)^2}=\sqrt{128}=11.31$	>	$d=\sqrt{(2-9)^2+(1-9)^2}=\sqrt{113}=10.63$

그래프에서 보이는 것처럼 초기 클러스터 0의 중심점은 (1, 1), 클러스터 1의 중심점은 (2, 1)으로 선택하였습니다. 이제 각 클러스터의 중심점과 전체 데이터 간의 거리를 각각 계산하여 클러스터로 할당해보겠습니다.

거리 계산을 한 표를 보면 각 데이터와 두 개의 클러스터 중심점과의 거리를 비교하였습니다. 즉, 1, 3번 데이터는 클러스터0의 중심점과 더 가깝고 나머지는 클러스터1의 중심점과 더 가까운 것으로 나타났습니다. 1, 3번 데이터는 클러스터 0에 할당되고 나머지 7개 데이터는 클러스터 1에 할당되었습니다. 그리고 클러스터 내 응집도가 최소화되는 새로운 중심 값을 계산해 보겠습니다.

(3) K 클러스터로의 모든 데이터 할당
(4) 각 클러스터 안의 모든 데이터들 간의 거리 평균값을 구하여 새로운 중심점으로 설정

데이터	x축	y축
1	1	1
2	2	1
3	1	2
4	2	2
5	3	3
6	8	8
7	8	9
8	9	8
9	9	9

클러스터	중심점	할당 데이터
0	(1, 1)	1, 3
1	(2, 1)	2, 4, 5, 6, 7, 8, 9

- 클러스터 0의 새로운 중심점

$$\frac{1+1}{2}=1,\ \frac{1+2}{2}=1.5 \rightarrow (1, 1.5)$$

- 클러스터 1의 새로운 중심점

$$\frac{2+2+3+8+8+9+9}{7}=5.85,\ \frac{1+2+3+8+9+8+9}{7}=5.71 \rightarrow (5.85, 5.71)$$

각 데이터 간의 거리 평균값을 구하기 위해 x, y값의 평균을 각각 구합니다. 즉, 클러스터 0의 x축의 평균값은 1, y축의 평균값은 1.5가 되어 새로운 중심점은 (1, 1.5)이 됩니다.

마찬가지로 동일하게 계산하여 클러스터 1의 새로운 중심점이 (5.85, 5.71)로 변경된 것을 확인할 수 있습니다. 각 클러스터의 중심점이 변경되었기 때문에, 다시 각 클러스터의 중심점과 전체 데이터 간의 거리 계산을 통해 클러스터 할당을 수행합니다.

(5) 각 클러스터의 중심점과의 거리 계산

데이터	클러스터 0의 중심점 (1, 1.5)과의 거리	대소 비교	클러스터 1의 중심점 (5.85, 5.71)과의 거리
1	$d = \sqrt{(1-1)^2+(1.5-1)^2} = 0.5$	<	$d = \sqrt{(5.85-1)^2+(5.71-1)^2} = 6.76$
2	$d = \sqrt{(1-2)^2+(1.5-1)^2} = 1.11$	<	$d = \sqrt{(5.85-2)^2+(5.71-1)^2} = 6.08$
3	$d = \sqrt{(1-1)^2+(1.5-2)^2} = 0.5$	<	$d = \sqrt{(5.85-1)^2+(5.71-2)^2} = 6.10$
4	$d = \sqrt{(1-2)^2+(1.5-2)^2} = 1.11$	<	$d = \sqrt{(5.85-2)^2+(5.71-2)^2} = 5.34$
5	$d = \sqrt{(1-3)^2+(1.5-3)^2} = 2.5$	<	$d = \sqrt{(5.85-3)^2+(5.71-3)^2} = 3.93$
6	$d = \sqrt{(1-8)^2+(1.5-8)^2} = 9.55$	>	$d = \sqrt{(5.85-8)^2+(5.71-8)^2} = 3.14$
7	$d = \sqrt{(1-8)^2+(1.5-9)^2} = 10.25$	>	$d = \sqrt{(5.85-8)^2+(5.71-8)^2} = 3.93$
8	$d = \sqrt{(1-9)^2+(1.5-8)^2} = 10.30$	>	$d = \sqrt{(5.85-9)^2+(5.71-8)^2} = 3.89$
9	$d = \sqrt{(1-9)^2+(1.5-9)^2} = 10.96$	>	$d = \sqrt{(5.85-9)^2+(5.71-9)^2} = 4.55$

새롭게 선택된 클러스터 0의 중심점 (1, 1.5), 클러스터 1의 중심점 (5.85, 5.71)과 전체 데이터 간의 거리를 계산합니다. 이번에는 데이터 1, 2, 3, 4, 5번이 클러스터 0의 중심점과 더 가깝고 6, 7, 8, 9번이 클러스터 1의 중심점과 더 가까운 것을 확인할 수 있습니다. 따라서 그래프에서 보는 것처럼 데이터들이 새롭게 클러스터링 됩니다. 그림 2는 새로운 클러스터의 두 번째 중심점과 데이터를 그래프로 표현했습니다.

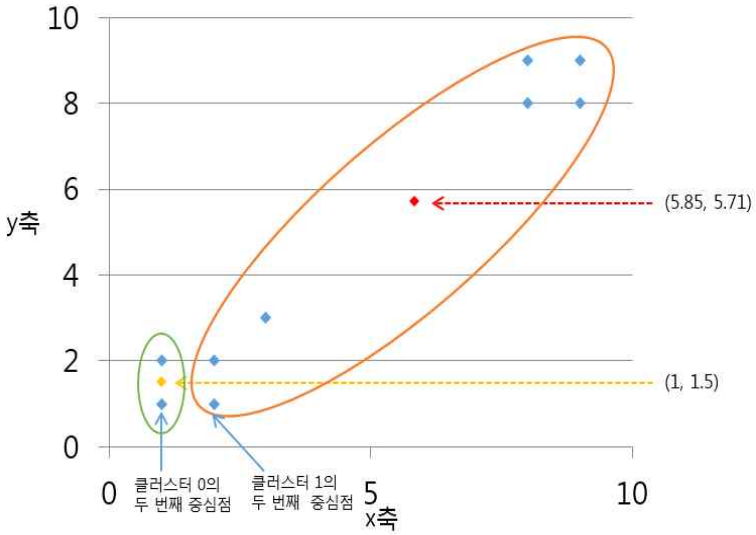


그림 2 두 번째 클러스터의 중심점과 데이터 그래프 표현

(6) K 클러스터로의 모든 데이터 할당

(7) 각 클러스터 안의 모든 데이터들 간의 거리 평균값을 구하여 새로운 중심점으로 설정

데이터	x축	y축
1	1	1
2	2	1
3	1	2
4	2	2
5	3	3
6	8	8
7	8	9
8	9	8
9	9	9

클러스터	중심점	할당 데이터
0	(1, 1.5)	1, 2, 3, 4, 5
1	(5.85, 5.71)	6, 7, 8, 9

- 클러스터 0의 새로운 중심점

$$\frac{1+2+1+2+3}{5}=1.8,\ \frac{1+1+2+2+3}{5}=1.8 \rightarrow (1.8, 1.8)$$

- 클러스터 1의 새로운 중심점

$$\frac{8+8+9+9}{4}=8.5,\ \frac{8+9+8+9}{4}=8.5 \rightarrow (8.5, 8.5)$$

이제 다시 각 클러스터의 중심점이 변하지 않는지 확인하기 위해 클러스터 내 데이터들의 평균값을 구해보겠습니다. 수식과 같이 클러스터 0 내의 데이터 평균값은 (1.8, 1.8), 클러스터 1 내의 데이터 평균값은 (8.5, 8.5)으로 계산됩니다.

(8) 각 클러스터의 중심점과의 거리 계산

데이터	클러스터 0의 중심점 (1.8, 1.8)과의 거리	대소 비교	클러스터 1의 중심점 (8.5, 8.5)과의 거리
1	$d = \sqrt{(1.8-1)^2+(1.8-1)^2} = 1.13$	<	$d = \sqrt{(8.5-1)^2+(8.5-1)^2} = 10.60$
2	$d = \sqrt{(1.8-2)^2+(1.8-1)^2} = 0.82$	<	$d = \sqrt{(8.5-2)^2+(8.5-1)^2} = 9.92$
3	$d = \sqrt{(1.8-1)^2+(1.8-2)^2} = 0.82$	<	$d = \sqrt{(8.5-1)^2+(8.5-2)^2} = 9.92$
4	$d = \sqrt{(1.8-2)^2+(1.8-2)^2} = 0.28$	<	$d = \sqrt{(8.5-2)^2+(8.5-2)^2} = 9.19$
5	$d = \sqrt{(1.8-3)^2+(1.8-3)^2} = 1.69$	<	$d = \sqrt{(8.5-3)^2+(8.5-3)^2} = 7.77$
6	$d = \sqrt{(1.8-8)^2+(1.8-8)^2} = 8.76$	>	$d = \sqrt{(8.5-8)^2+(8.5-8)^2} = 0.70$
7	$d = \sqrt{(1.8-8)^2+(1.8-9)^2} = 9.50$	>	$d = \sqrt{(8.5-8)^2+(8.5-8)^2} = 0.70$
8	$d = \sqrt{(1.8-9)^2+(1.8-8)^2} = 9.50$	>	$d = \sqrt{(8.5-9)^2+(8.5-8)^2} = 0.70$
9	$d = \sqrt{(1.8-9)^2+(1.8-9)^2} = 10.18$	>	$d = \sqrt{(8.5-9)^2+(8.5-9)^2} = 0.70$

새로운 평균값은 중심점과 다르기 때문에 새로운 중심점이 선택되고, 다시 앞의 과정을 반복합니다. 다시 새로운 클러스터들의 중심점과 전체 데이터 간의 거리를 계산합니다. 각 중심점과의 데이터들 간의 거리의 비교가 앞의 결과와 동일한 것을 확인할 수 있습니다. 앞과 동일하게 1, 2, 3, 4, 5번 데이터는 클러스터 0에 할당되고 6, 7, 8, 9번 데이터는 클러스터 1에 할당되었습니다. 그림 3은 세 번째 클러스터의 중심점과 데이터를 그래프로 표현했습니다.

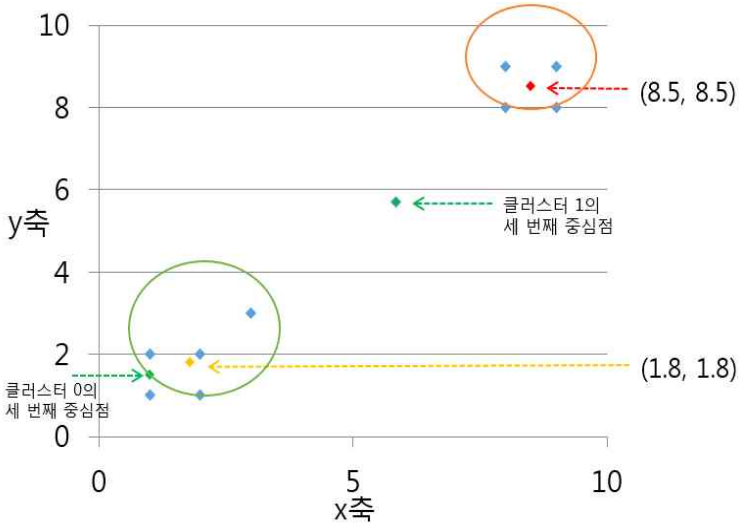


그림 3 세 번째 클러스터의 중심점과 데이터 그래프 표현

(9) K 클러스터로의 모든 데이터 할당

(10) 각 클러스터 안의 모든 데이터들 간의 거리 평균값을 구하여 새로운 중심점 설정

데이터	x축	y축
1	1	1
2	2	1
3	1	2
4	2	2
5	3	3
6	8	8
7	8	9
8	9	8
9	9	9

클러스터	중심점	할당 데이터
0	(1.8, 1.8)	1, 2, 3, 4, 5
1	(8.5, 8.5)	6, 7, 8, 9

- 클러스터 0의 새로운 중심점

$$\frac{1+2+1+2+3}{5}=1.8,\ \frac{1+1+2+2+3}{5}=1.8\rightarrow(1.8,1.8)$$

- 클러스터 1의 새로운 중심점

$$\frac{8+8+9+9}{4}=8.5,\ \frac{8+9+8+9}{4}=8.5\rightarrow(8.5,8.5)$$

다시 클러스터 내 데이터의 평균값을 구해보겠습니다. 새롭게 구한 평균값이 클러스터 0은 (1.8, 1.8), 클러스터1은 (8.5, 8.5)로 기존의 중심점과 동일하기 때문에 K-means 알고리즘의 수행은 종료됩니다. 그림 4는 네 번째 클러스터의 중심점과 데이터를 그래프로 표현했습니다.

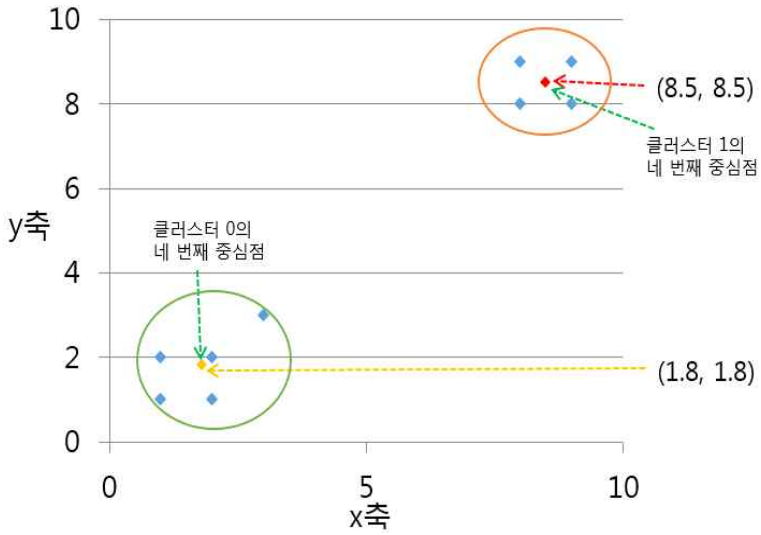


그림 4 네 번째 클러스터의 중심점과 데이터 그래프 표현

(11) 클러스터링 최종 결과

최종적으로 클러스터 0의 중심점은 (1, 1)에서 (1.8, 1.8)로, 클러스터 1의 중심점은 (2, 1)에서 (8.5, 8.5)로 이동하였습니다. 그리고 9개의 데이터가 각각 5개, 4개씩 클러스터 0, 클러스터 1에 할당된 것을 확인할 수 있었습니다. 그림 5는 클러스터링 최종 결과입니다.

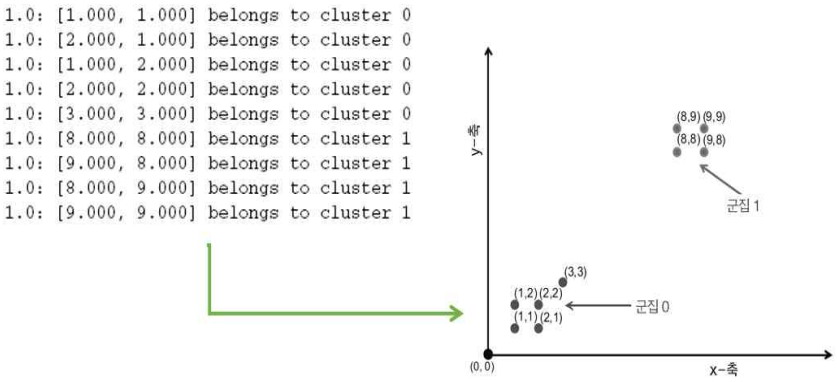


그림 5 최종 클러스터링 결과

지금까지 간단한 예제를 통해 K-means 알고리즘의 로직을 살펴보았습니다.

이제 scikit-learn 라이브러리를 이용해 K-means 클러스터링을 실습해 볼 텐데요. 실습을 하기 전에, scikit-learn 라이브러리에 대해 간단히 알아보겠습니다.

1.3 Scikit-learn(Sklearn) 패키지 소개

1.3.1 Scikit-learn 패키지

- Python에서 Machine Learning(머신러닝)을 위한 데이터 및 함수를 제공하는 대표적인 패키지
- Machine Learning의 Classification(분류), Regression(회귀), Clustering(군집) 등 다양한 알고리즘들이 제공
- 공식 사이트 : http://scikit-learn.org/stabel/

Scikit-learn 패키지는 머신러닝을 위한 예제 데이터와 관련 함수를 제공하는 라이브러리로, 분류, 회귀, 군집 등 다양한 알고리즘을 제공합니다.

1.3.2 Scikit-learn 대표 기능 및 함수 설명

(1) 예제 데이터 셋 로드 함수
- ex) sklearn.datasets.load_iris(), sklearn.datasets.load_flights() 등

(2) 학습 및 예측
- ex) sklearn.svm.SVC.fit() : 모델에 맞게 학습하는 함수
- ex) sklearn.svm.SVC.predict() : 새로운 값을 예측하는 함수, 즉 새로운 데이터의 라벨을 예측하는 함수

(3) 파라미터 재설정 및 재학습 함수
- ex) sklearn.svm.SVC.set_params() : 모델의 파라미터를 변경하는 함수
- ex) sklearn.svm.SVC.set_params.fit() : 모델의 파라미터를 변경하고 다시 모델에 맞게 학습하는 함수

Scikit-learn의 대표 기능들을 살펴보겠습니다. Scikit-learn에서는 머신 러닝 모델을 학습시킬 수 있는 다양한 데이터 셋을 제공합니다. 이는 scikit-learn의 서브 패키지 datasets에서 load_iris()와 같은 함수를 통해 가져올 수 있습니다.

그리고 데이터로 모델을 학습시키기 위해서는 fit()함수를, 새로운 데이터를 예측하기 위해서는 predict() 함수를 사용합니다. 이 밖에도 set_params(), set_params.fit()함수를 사용해 파라미터 재설정 및 재학습을 할 수 있습니다.

1.4 K-means 실습

1.4.1 예제 1 : scikit-learn 패키지를 이용한 클러스터링 1

1.4.1.1 Python 패키지 가져오기 및 matplotlib 출력 옵션 설정

첫 번째 줄에서 sklearn.cluster 패키지의 Kmeans를 import합니다. 그리고 두 번째 줄에서 샘플 데이터 생성을 위해 사용할 make_blobs를 불러옵니다. 네 번째 줄에서는 수학 계산을 위해 사용할 numpy 패키지를 np라는 이름으로 import하고 다섯 번째 줄에서는 데이터를 시각화하기 위해 matplotlib.pyplot 패키지를 plt 라는 이름으로 import합니다.

그리고 일곱 번째 줄에서 %기호로 시작하는 코드는 ipython에서 제공하는 magic function으로, matplotlib의 시각화 결과를 ipython notebook 에서 출력할 수 있도록 설정하겠습니다. 그림 6은 패키지 import 및 matplotlib 출력 옵션 설정에 대한 내용입니다.

```
1 from sklearn.cluster import KMeans
2 from sklearn.datasets import make_blobs
3
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 %matplotlib inline
```

그림 6 패키지 import 및 matplotlib 출력 옵션 설정

1.4.1.2 클러스터링 샘플 데이터 생성

이제 클러스터링을 할 샘플 데이터를 만들어보겠습니다. make_blobs는 클러스터링 모델을 시험할 때 원하는 특성을 가진 데이터를 만들기 위해 사용하는 함수입니다.

먼저 두 번째 줄에서 numpy.random 패키지의 seed() 함수를 호출해 시드 값을 설정하는 코드입니다. numpy.random은 랜덤 값을 생성하는 여러 함수들을 제공하는 패키지로, 랜덤 값은 시드 값을 바탕으로 생성됩니다. 그렇다면 시드 값이란 무엇일까요? 간단히 말씀드리면, 랜덤 값들이 생성될 때 사용되는 초기값으로, 만약 시드 값이 설정되어 있지 않다면 랜덤 값을 생성할 때마다 다른 값들이 생성되게 됩니다. 따라서 우리는 동일한 랜덤 값들을 생성

시키기 위하여 시드 값을 설정합니다.

이 시드 값은 make_blobs() 함수가 데이터를 생성할 때 영향을 미치게 됩니다. 시드 값을 설정했으니 이제 make_blobs()함수로 우리가 원하는 특성의 데이터를 생성해보겠습니다.

만들 데이터는 총 2000개의 3개의 중심점을 기반으로 한, 클러스터의 표준편차 값이 0.7인 데이터로 설정해보겠습니다. 이를 위해 먼저 네 번째 줄에서 데이터들이 군집하여 생성될 중심점 세 개 값을 리스트 형태로 설정합니다. 이 값은 make_blobs 함수의 centers 파라미터에 설정됩니다.

즉, [1, 1], [0, 0], [2, -1] 세 중심축을 기준으로 하는 데이터를 생성할 수 있습니다. 열 번째 줄에서 make_blobs() 함수를 호출해 데이터를 생성해보겠습니다. make_blobs() 함수의 n_samples 파라미터에 생성하려는 데이터의 개수 2000을 넣고 centers에는 위에서 생성한 변수 centers를 넣어줍니다.

그리고 cluster_std 파라미터를 통해 각 군집의 표준편차를 정해줍니다. 표준편차란 데이터의 산포도를 나타내는 수치로 표준편차가 클수록 중심점으로부터 데이터들의 거리가 멀어집니다. 그리고 make_blobs() 함수 호출 시 반환되는 값을 각각 data와 labels_true 변수에 할당하겠습니다. 그림 7은 클러스터링에 사용할 샘플 데이터 생성에 대한 내용입니다.

```
1  # 시드값 설정
2  np.random.seed(0)
3
4  centers = [[1, 1], [0, 0], [2, -1]]
5  n_clusters = len(centers) # 3개
6
7  # 데이터 생성
8  # centers : the number of centers to generate
9  # cluster_std : fixed center locations
10 data, labels_true = make_blobs(n_samples = 2000, centers = centers,
11                                cluster_std = 0.7)
```

그림 7 클러스터링 샘플 데이터 생성

1.4.1.3 데이터 및 라벨 확인

print()문에 data를 넣어 값을 출력해보면 2000x2의 배열 형태로 2개의 feature 값을 가진 2000개의 샘플 데이터가 생성되었음을 확인할 수 있습니

다. 그리고 labels_true 변수를 print()문으로 출력해 봅니다.

labels_true 변수는 데이터의 실제 라벨 값으로 네 번째 줄에서 numpy의 unique()함수를 사용하여 labels_true 변수에서 중복을 제거한 값을 출력해보았습니다. 출력된 것처럼 데이터의 라벨은 0, 1, 2 세 개로 이루어진 것을 확인할 수 있습니다. 즉, 우리가 세 개의 중심점을 기준으로 생성한 데이터들이 각각 0, 1, 2 클러스터에 소속된 것을 알 수 있습니다. 그림 8은 샘플 데이터 및 라벨을 확인하는 내용입니다.

```
1 print(data)
2 print()
3 print(labels_true)
4 print(np.unique(labels_true))
```

```
[[ 2.88735684  0.94825273]
 [ 0.00712986  1.53880744]
 [ 0.3264657  -0.06607475]
 ...,
 [ 0.53901292  0.64003622]
 [ 1.65065358  1.40755721]
 [ 0.74131908 -0.71579507]]

[0 1 1 ..., 0 0 1]
[0 1 2]
```

그림 8 샘플 데이터 및 라벨 확인

1.4.1.4 2차원 평면에 데이터 표현

샘플 데이터를 이용하여 클러스터링 모델을 생성하기 전에 먼저 생성한 데이터들이 어떻게 분포되어있는지 matplotlib 라이브러리를 사용해 산포도를 나타내어볼까요? matplotlib 라이브러리를 이용해 데이터의 산포도를 나타내는 방법은 간단합니다.

먼저 첫 번째 줄에서 pyplot의 figure()함수를 호출하여 그림의 크기를 가로 15, 세로 10으로 설정하겠습니다. 그 다음 pyplot의 scatter() 함수 안에 산포도로 나타낼 데이터의 x축 값, y축 값을 각각 파라미터로 넣어주어 호출합니다. 여기서 X축 값은 data 변수의 첫 번째 feature 값을 , Y축 값은 두 번째

feature 값을 사용했습니다.

즉, data 변수의 첫 번째 열 데이터, data 변수의 두 번째 열 데이터를 각각 넣어 scatter()함수를 호출하면 그림과 같이 데이터가 분포되어있는 것을 확인할 수 있습니다. 그림 9는 샘플 데이터의 분포를 산포도로 나타낸 내용입니다.

```
1 plt.figure(figsize=(15,10))
2 plt.scatter(data[:,0], data[:,1])
```

```
<matplotlib.collections.PathCollection at 0xcf2c400>
```

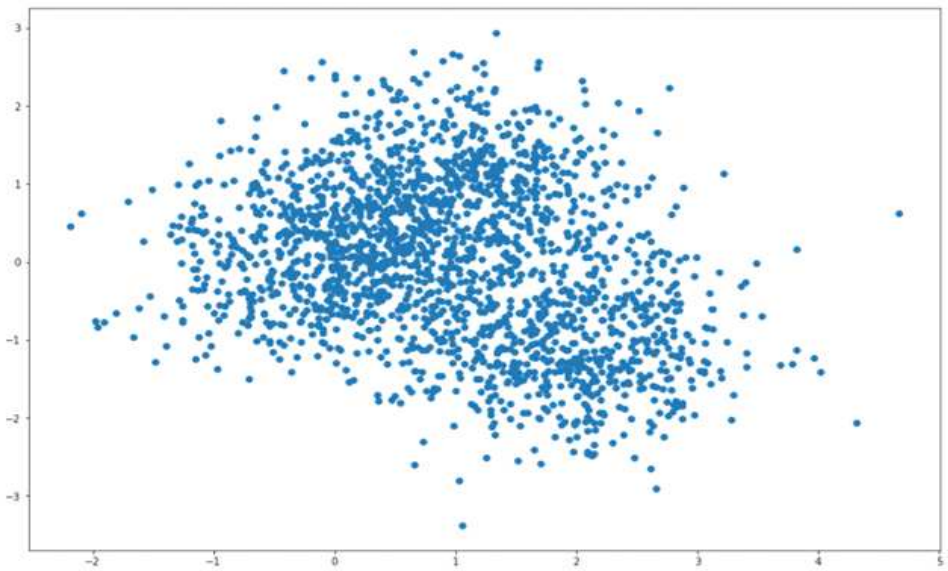


그림 9 샘플 데이터의 분포를 산포도로 표현

1.4.1.5 모델 학습

이제 K-means 클러스터링 모델을 만들어 샘플 데이터로 학습시켜보겠습니다. 두 번째 줄을 보면 init, n_clusters, n_init 세 개의 파라미터를 사용하여 Kmeans() 함수를 호출해 K-means 모델을 생성했습니다. init은 초기 K값을 찾는 함수를 설정하는 파라미터로 기본 함수인 k-means++ 함수를 사용하도록 하겠습니다.

그리고 n_clusters 파라미터는 클러스터링 할 클러스터의 개수인 3으로 설정합니다. n_init 파라미터는 초기 중심점 선택 시 최상의 결과를 얻기 위해 몇 번 초기 값을 변경할 것인지를 설정하는 횟수입니다. 파라미터를 설정한

후 K-means 초기 모델을 estimator 변수에 할당한 후 fit()함수를 사용해 data를 클러스터링 합니다. 그림 10은 클러스터링 모델을 생성하고 샘플 데이터 학습에 대한 내용입니다.

```
# compute clustering with KMeans
estimator = KMeans(init = 'k-means++', n_clusters = 3, n_init = 10)
estimator.fit(data)
```

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

그림 10 클러스터링 모델 생성 및 샘플 데이터 학습

1.4.1.6 클러스터링 결과 확인

이제 클러스터링 결과를 확인해보겠습니다. 앞에서 fit()함수를 호출하게 되면 estimator 변수는 labels_ 라는 클래스 변수에 클러스터 결과를 저장합니다. 이 값을 labels_predict 변수에 저장하고 numpy의 unique 함수를 사용하여 중복 제거 후 값을 출력해보면 클러스터 0, 1, 2 세 개로 클러스터링 된 것을 확인할 수 있습니다. 그림 11은 학습한 모델로 예측한 라벨 데이터입니다.

```
labels_predict = k_means.labels_
labels_predict
```

```
array([0, 0, 2, ..., 0, 0, 2])
```

그림 11 학습한 모델로 예측한 라벨 데이터

이제 결과로 나온 클러스터 라벨에 따라 데이터를 각각 다른 색으로 표현해 산포도를 그려보겠습니다. 산포도에 색을 입히기 위해서 matplotlib에서 제공하는 colormap을 사용할건데, 기본적으로 colormap 변수는 0에서 1사이의 값을 인자로 받아 그에 상응하는 RGBA값을 반환합니다.

따라서 클러스터마다 다른 색으로 데이터를 나타내기 위해 colormap 변수 안에 클러스터 라벨 값을 넣어 RGBA 값을 받아올 것입니다. 여기서 클러스터 라벨 값이 0에서 1사이의 값으로 변경될 필요가 있기 때문에 이 값을 정규화 방법을 사용해 변환시키겠습니다.

먼저 두 번째 줄에서 colormap 코드 중 하나인 ‘jet’를 pyplot.cm.get_cmap()

함수로 가져와 cm 변수에 저장합니다. 그리고 세 번째 줄과 네 번째 줄을 통해 정규화 작업을 합니다. 먼저 labels_predict 변수에서 최솟값을 뺀 값을 scaled_labels 변수에 저장합니다.

그리고 scaled_labels 변수를 최댓값에서 최솟값을 뺀 값으로 나누어 다시 scaled_labels 변수에 저장합니다. 이제 정규화된 scaled_labels 변수의 값을 확인해보겠습니다. 출력된 결과 값을 확인하면 클러스터 라벨 값이 0, 1, 2에서 0, 0.5, 1로 정규화된 것을 확인할 수 있을 겁니다. 그림 12는 예측 라벨 데이터를 정규화하는 내용입니다.

```
# normalize into [0,1]
cm = plt.cm.get_cmap('jet')
scaled_labels = (labels_predict - np.min(labels_predict))
scaled_labels = scaled_labels /(np.max(labels_predict) - np.min(labels_predict))
np.unique(scaled_labels)
```

```
array([ 0. ,  0.5,  1. ])
```

그림 12 예측 라벨 데이터 정규화

1.4.1.7 클러스터링 결과 확인

이제 pyplot의 scatter() 함수를 사용해 클러스터링 결과를 확인하도록 하겠습니다. 먼저 첫 번째 줄에서 그림의 크기를 지정한 후 두 번째 줄에서 scatter() 함수를 호출합니다. 앞에서는 파라미터로 데이터만 설정했다면 이번에는 산포도에 색상을 입히기 위해 c 파라미터를 사용하였습니다.

c 파라미터에는 cm colormap 변수에 scaled_labels 변수 값을 넣어 생성된 클러스터 라벨 값에 따른 RGBA값이 들어갑니다. 산포도 결과를 확인해보면 유사한 데이터끼리 클러스터링이 잘 되었다는 것을 확인할 수 있습니다. 그림 13은 클러스터링 결과를 산포도로 표현하는 내용입니다.

```
plt.figure(figsize=(15,10))
plt.scatter(data[:,0], data[:,1], c = cm(scaled_labels))
```

```
<matplotlib.collections.PathCollection at 0xd8ea048>
```

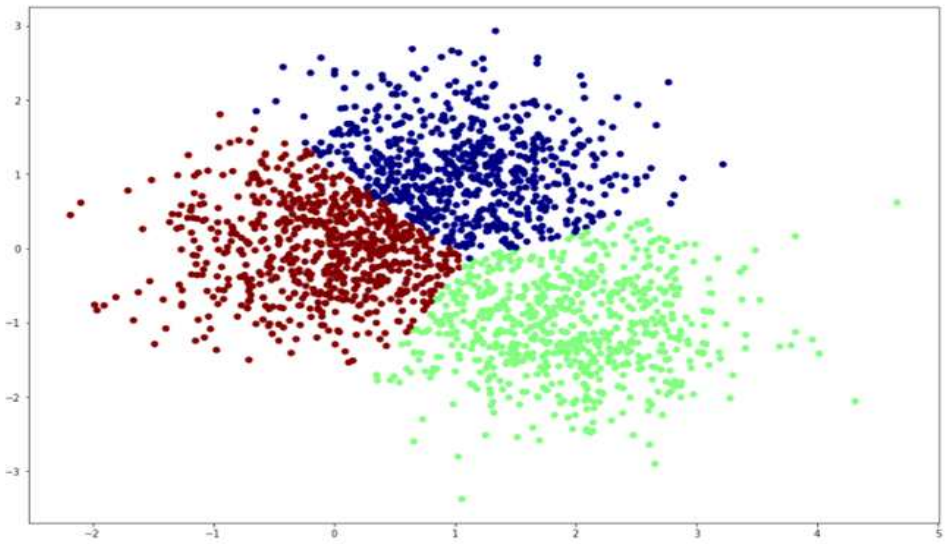


그림 13 클러스터링 결과

이번에는 조금 더 나아가 K-means 모델의 초기 k 값을 선정하는 init 함수를 달리하여 결과를 비교하고, 몇 가지 지표를 통해 클러스터링 성능 평가를 해보겠습니다.

1.4.2 예제 2 : scikit-learn 패키지를 이용한 클러스터링 2

1.4.2.1 Python 패키지 가져오기 및 난수로 이루어진 샘플 데이터 생성을 위한 seed 값 설정

먼저 python 패키지를 import하겠습니다. 첫 번째 줄에서 데이터를 가져오기 위한 load_digits() 함수를 import하고 두 번째 줄에서 데이터 전처리를 위한 scale() 함수를, 세 번째 줄에서 pandas 라이브러리를 import합니다. 그림 14는 패키지 import 및 seed 설정에 대한 내용입니다.

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
import pandas as pd

np.random.seed(60)
```

그림 14 패키지 import 및 seed 설정

1.4.2.2 Digits 데이터 가져오기 및 데이터, 라벨 확인

load_digits() 함수로 가져온 데이터는 0에서 9까지의 정수로 이루어진 데이터 셋이며 우리는 이 데이터를 데이터 전처리 중 하나인 스케일을 통해 값의 규모를 조정해보겠습니다. 스케일은 뒤에서 자세히 알아보겠습니다.

먼저 첫 번째 줄에서 load_digits() 함수로 가져온 digits 데이터를 digits 변수에 할당합니다. 그리고 digits 변수의 data 값을 data 변수에 저장합니다. 여기서 data 값은 digits 데이터의 feature 값들을 의미합니다. 이제 scale() 함수를 통해 데이터를 전처리를 해보겠습니다. 스케일링이란 무엇일까요? 데이터는 여러 feature 값들을 가지고 있을 수 있는데, 이 feature들은 각각 값의 범위가 가지각색일 수 있습니다.

예를 들어 사람의 나이, 몸무게, 월급 등으로 feature가 이루어져있다면 그 값의 범위가 많이 다르겠죠. 이 때 이처럼 feature의 값의 범위에 차이가 많은 데이터를 사용하면 모델이 잘 만들어지지 않을 수 있습니다. 즉, 성능에 영향을 미치게 됩니다.

왜냐하면 머신러닝의 클러스터링, 분류 등의 알고리즘은 데이터 간의 거리를 기반으로 만들어지는 경우가 많기 때문에, 특정 feature 값의 범위가 큰 경우에 그 feature 가 다른 feature들보다 모델에 더 많은 영향을 주기 때문이죠. 그래서 모든 feature가 모델 생성 및 학습에 최대한 동일한 기여를 하게 하기 위하여, 모델 생성 전 스케일링 같은 데이터 전처리 과정을 필수적으로 거칩니다.

scikit-learn 라이브러리에서 제공하는 스케일링 함수인 scale()은 각 데이터의 feature 값의 분포를 평균 0, 분산 1로 만듭니다. 이제 데이터 스케일을 한 후 전후를 출력하여 비교해보겠습니다.

네 번째 줄에서 스케일링 전 데이터를 출력하고 여섯 번째 줄에서 scale() 함수를 사용하여 스케일링한 데이터를 다시 data 변수에 저장하여 출력합니다. 전후를 비교해보면 각 데이터의 feature 값의 범위가 비슷하게 줄어든 것을 확인할 수 있습니다. 이제 생성된 데이터의 형태와 라벨 값들을 확인해보겠습니다. 그림 15는 digits 데이터를 스케일링 하는 내용입니다.

```
digits = load_digits()
data = digits.data
print("< Before scaling >")
print(data)
print("< After scaling >")
data = scale(data)
print(data)
```

```
< Before scaling >
[[ 0.  0.  5. ...,  0.  0.  0.]
 [ 0.  0.  0. ..., 10.  0.  0.]
 [ 0.  0.  0. ..., 16.  9.  0.]
 ...,
 [ 0.  0.  1. ...,  6.  0.  0.]
 [ 0.  0.  2. ..., 12.  0.  0.]
 [ 0.  0. 10. ..., 12.  1.  0.]]
< After scaling >
[[ 0.         -0.33501649 -0.04308102 ..., -1.14664746 -0.5056698
  -0.19600752]
 [ 0.         -0.33501649 -1.09493684 ...,  0.54856067 -0.5056698
  -0.19600752]
 [ 0.         -0.33501649 -1.09493684 ...,  1.56568555  1.6951369
  -0.19600752]
 ...,
 [ 0.         -0.33501649 -0.88456568 ..., -0.12952258 -0.5056698
  -0.19600752]
 [ 0.         -0.33501649 -0.67419451 ...,  0.8876023  -0.5056698
  -0.19600752]
 [ 0.         -0.33501649  1.00877481 ...,  0.8876023  -0.26113572
  -0.19600752]]
```

그림 15 digits 데이터 스케일링

1.4.2.3 데이터 개수 및 feature, 라벨 개수 확인

세 번째 줄에서는 numpy의 unique() 함수를 사용하여 실제 라벨 값들을 clusters 변수에 저장합니다. 네 번째 줄에서는 클러스터의 개수를 len() 함수를 이용해 n_clusters 변수에 할당합니다. 그리고 생성한 변수들의 값을 출력해 보겠습니다.

출력된 결과처럼 총 1797개의 데이터는 각 64개의 feature들을 갖습니다. 또한 데이터는 0부터 9까지 10개의 클러스터로 나눠져 있는 것을 확인할 수 있습니다. 그림 16은 데이터, feature와 라벨 개수 확인하는 내용입니다.

```
labels_true = digits.target
n_samples, n_features = data.shape
clusters = np.unique(labels_true)
n_clusters = len(clusters)

print("n_samples : " + str(n_samples))
print("n_features : " + str(n_features))
print("n_clusters : " + str(n_clusters))
print("clusters : " + str(clusters))
```

```
n_samples : 1797
n_features : 64
n_clusters : 10
clusters : [0 1 2 3 4 5 6 7 8 9]
```

그림 16 데이터, feature와 라벨 확인

1.4.2.4 초기 K값 설정을 위한 함수 두 개를 각각 사용해 모델 학습

우리는 이제 K-means()함수를 사용해 모델을 만들고 feature 값을 넣어 데이터를 클러스터링 해보겠습니다. 이번에는 K 값 설정을 위한 init 함수의 성능을 비교하기 위해 k-means++ 함수와 random 함수 각각을 사용한 두 개의 모델을 생성합니다.

먼저 첫 번째 줄에서 estimator1 변수에 k-means++ 함수를 사용하여 생성한 K-means 모델을 저장합니다. n_clusters 파라미터에는 앞에서 생성한 n_cluster 변수를, n_init 파라미터에는 10을 넣어 호출하였습니다. 그리고 두 번째 줄에서 feature 값인 data 변수를 fit()함수에 넣어 클러스터링을 합니다.

네 번째 줄에서는 'random'함수를 사용한 kmeans 모델을 estimator2에 저장합니다. 그리고 마찬가지로 fit()함수로 클러스터링을 하였습니다. 일곱 번째, 여덟 번째 줄에서는 두 모델을 사용해 클러스터링한 데이터의 클러스터 라

벨 값을 각각 labels_predict1, labels_predict2 변수에 저장하였습니다. 그림 17은 각각 다른 함수를 사용해 생성한 두 개의 K-means 모델을 학습시키는 내용입니다.

```
estimator1 = KMeans(init = 'k-means++', n_clusters = n_clusters, n_init = 10)
estimator1.fit(data)

estimator2 = KMeans(init = 'random', n_clusters = n_clusters, n_init = 10)
estimator2.fit(data)

labels_predict1 = estimator1.labels_
labels_predict2 = estimator2.labels_
```

그림 17 두 개의 각각 다른 함수 K-means 모델 학습시키기

1.4.2.5 각 함수에 대해 클러스터링 성능평가 및 비교

이제 이 예측한 클러스터 라벨 값을 이용해 모델의 클러스터링 성능평가를 해보겠습니다. 그 전에 클러스터링 성능평가를 위한 대표적인 지표 세 가지를 짚고 넘어가겠습니다.

먼저 첫 번째로, 동질성으로 해석할 수 있는 homogeneity 값은 모델을 통해 클러스터링 된 각 클러스터 안의 데이터들이 실제 하나의 같은 클러스터 라벨을 갖는가 하는 것을 평가하는 지표입니다. 조금 다르게 completeness 지표는 실제 같은 클러스터 라벨을 가지고 있는 데이터들이 모델을 통해 하나의 같은 클러스터로 클러스터링 되었는가를 평가합니다.

그리고 v-measure 값은 이 두 값을 모두 사용하여 수식적으로 계산한 지표 값입니다. 이 대표적인 세 지표 값을 통해 클러스터링 성능을 평가해보겠습니다. 먼저 변수 labels_predict1과 labels_predict2를 실제 라벨 값인 변수 labels_true와 비교하여 세 가지 지표로 성능 평가를 해보겠습니다.

metrics 패키지의 homogeneity_score(), completeness_score(), v_measure_score()는 모두 첫 번째 인자로 실제 라벨 값, 두 번째 인자로 예측한 라벨 값을 파라미터로 갖습니다. 두 모델을 비교하기 위해 네 번째, 열한 번째 줄에서 print()문 안에 init 함수와 함께 모델을 비교하는 문자열을 출력했습니다. 그리고 각각 밑에 세 가지 평가 지표 값을 출력했습니다.

지표 값을 출력하는 print()문 안의 문자열 안에는 모두 %(percent) 기호가 있는데, 이는 뒤의 %로 시작하는 값을 이 위치에 사용자가 원하는 형태의 문자열로 포맷팅하여 출력하는 연산자입니다.

여기서 % 기호 뒤의 .3f는 값은 소수점 셋 째 자리까지 표시한 후 문자열로 나타내라는 것을 의미합니다. 즉, 출력된 결과에서 확인할 수 있는 것과 같이 모든 지표 값은 소수점 셋째 자리까지 표현된 것을 확인할 수 있습니다. 그림 18은 클러스터링 성능평가 및 비교에 대한 내용입니다.

```
from sklearn import metrics

print("< clustering performance evaluation >₩n")
print("1. clustering with initializing first centroids of clusters with k-means+
print('homogenity score : %.3f'
      %(metrics.homogeneity_score(labels_true, labels_predict1)))
print('completeness score : %.3f'
      %(metrics.completeness_score(labels_true, labels_predict1)))
print('v-measure score : %.3f ₩n'
     %(metrics.v_measure_score(labels_true, labels_predict1)))
print("2. clustering with initializing first centroids of clusters randomly ")
print('homogenity score : %.3f'
      %(metrics.homogeneity_score(labels_true, labels_predict2)))
print('completeness score : %.3f'
      %(metrics.completeness_score(labels_true, labels_predict2)))
print('v-measure score : %.3f ₩n'
     %(metrics.v_measure_score(labels_true, labels_predict2)))
```

그림 18 클러스터링 성능평가 및 비교

이제 두 모델의 성능 평가 값을 비교해보겠습니다. 그림 19는 클러스터링 성능평가 및 비교 결과입니다.

```
< clustering performance evaluation >

1. clustering with initializing first centroids of clusters with k-means++ function
homogenity score : 0.669
completeness score : 0.711
v-measure score : 0.689

2. clustering with initializing first centroids of clusters randomly
homogenity score : 0.611
completeness score : 0.658
v-measure score : 0.634
```

그림 19 클러스터링 성능평가 및 비교 결과

세 개의 값 모두 K-means++ 함수로 초기 K값을 정한 estimator1 모델의 값이 더 높은 성능을 낸 것을 확인할 수 있습니다. 지금까지 클러스터링의 대표적인 알고리즘인 K-means 모델로 클러스터링을 하는 방법과 클러스터링 성능 평가에 대해 간단히 알아보았습니다.

2. K-Nearest Neighbors(KNN)

2.1 K-Nearest Neighbors (KNN) 알고리즘이란?

2.1.1 KNN 알고리즘 소개

(1) 머신러닝의 지도 학습의 한 종류인 분류 문제를 해결하는 알고리즘 중 하나

(2) 주어진 데이터로부터 거리가 가까운 K개의 다른 데이터들의 라벨 중 가장 많은 비율을 차지하는 라벨을 참조하여 분류하는 알고리즘

(3) 주로 거리를 측정할 때 유클리드 거리 측정법을 사용

- 지도 학습(Supervised Learning)
  - 훈련 데이터(Training Data)로부터 하나의 함수를 추론하는 방법 중 하나로, 라벨(Label)이 있는 훈련 데이터를 학습하여 함수를 추론하는 것으로 일반적으로 분류(Classification), 회귀(Regression) 문제를 위해 사용된다.

- 분류(Classification)
  - 지도 학습의 일종으로, 라벨이 있는 데이터를 학습하여 새로운 데이터가 들어왔을 때 학습한 모델을 이용하여 라벨을 붙이는 것, 즉 카테고리를 나누는 것

(4) 알고리즘이 간단하여 구현하기 쉽고 정확도가 좋은 편에 속함

(5) 사례 기반 알고리즘(Instance-based Algorithms)의 한 종류로 학습 과정이 따로 없이 각 데이터를 분류할 때마다 전체 데이터를 탐색해야 하기 때문에, 특히 데이터의 양이 많아지면 속도가 상당히 느려짐

(6) Python에서는 대표적으로 Scikit-learn 패키지에서 KNN 알고리즘을 이용한 분류 문제 해결을 위한 함수를 제공

- 사례 기반 알고리즘(Instance-based Algorithms)
  - 훈련 데이터로 이루어진 데이터베이스를 만들고, 새로운 데이터를 데이터베이스의 데이터들과의 유사도를 측정하는 방식으로 비교하여 예측을 수행한다. 대표적 사례 기반 알고리즘은 KNN, LVQ 등이 있다.

KNN은 머신러닝의 지도 학습의 한 종류인 분류 문제를 해결하는 알고리즘 중 하나입니다. 지도 학습이란 라벨이 있는 데이터를 학습하여 함수를 추론하는 것으로, 학습에 사용되는 데이터의 결과가 정해져 있는 경우에 사용합니다. 즉, 지도 학습의 대표적인 예인 분류 문제는 라벨이 있는 데이터를 학습하여 새로운 데이터가 들어왔을 때 학습된 모델을 데이터를 라벨을 붙여 분류하는 것을 의미합니다.

이제 KNN 알고리즘에 대해 자세히 살펴보겠습니다. KNN 알고리즘은 주어진 데이터로부터 가장 거리가 가까운 K개의 다른 데이터를 참조하여 라벨링하는 알고리즘으로, K개 데이터들의 라벨들 중 가장 많은 비율을 차지하는 라벨로 분류하게 됩니다. 여기서 데이터 간의 거리는 일반적으로 유클리드 거리 측정법을 사용합니다.

KNN 알고리즘은 이처럼 간단하게 구현할 수 있는 알고리즘이지만, 사례 기반 알고리즘의 한 종류로 데이터의 양이 많아지면 수행 속도가 느려질 수 있습니다. 사례 기반 알고리즘이란 간단히 설명해 학습 과정이 따로 없이 각 데이터 분류 시 전체 데이터를 계속 탐색해야 하는 알고리즘입니다. 즉, 데이터의 개수가 많아질수록 알고리즘의 수행 시간 또한 길어지게 되겠죠? 유클리드 거리 측정법에 대해 간단히 알아보겠습니다.

2.1.2 유클리드 거리 측정법

$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2}$$

- 유클리드 거리가 큰 값을 가지면 사용자 간의 거리가 멀다는 의미이기 때문에 사용자 간 유사성이 떨어짐을 의미함

이는 다음과 같이 각 벡터 간의 차를 제곱하여 모두 더한 후 루트를 씌워 계산합니다. 여기서 벡터란 데이터의 각 feature 들을 의미합니다. 이제 KNN 알고리즘의 수행 과정을 네 단계로 그림을 통해 쉽게 이해해볼까요?

2.1.3 KNN 알고리즘 수행 과정

먼저 첫 번째 그림에서 분류할 데이터를 2차원 그래프로 확인해보겠습니다. 현재 노란색, 초록색, 빨간색으로 라벨을 구분한 8개의 샘플 데이터가 존

재하고, 검은색 점은 이를 참조해 이제부터 분류하고자 하는 테스트 데이터입니다.

두 번째로, 검은색 데이터와 가장 가까운 K개의 데이터를 찾아볼 텐데요, K 값을 임의로 4로 지정하겠습니다. 즉, 분류하고자 하는 데이터와 가장 가까운 4개의 데이터를 찾아보겠습니다. 두 번째 그림을 보면, 검은색 데이터와 나머지 8개 데이터 간의 거리를 각각 구합니다. 이 때 거리는 유클리드 거리 측정법을 사용하여 계산합니다. 그리고 그래프 위에 가장 가까운 네 개의 데이터와의 거리를 표시하였습니다.

세 번째로, 4개의 데이터를 확인해보겠습니다. 그림에서 보면 4개의 데이터는 거리가 가까운 순으로 1st, 2nd, 3rd, 4th로 이름을 붙여 검은색 데이터의 이웃 데이터로 명명했습니다. 참고로 4개의 데이터를 거리에 따라 순서를 매긴 것은 거리가 가까운 데이터에 더 많은 가중치를 주어 KNN 알고리즘을 수행하는 경우가 있기 때문입니다. 이는 뒤에서 실습을 통해 확인해보겠습니다.

이제 네 번째 그림처럼 각 데이터의 라벨 값을 확인하겠습니다. 결과적으로 노란색 클래스로 분류된 데이터가 2개, 초록색 클래스 데이터가 1개, 빨간색 클래스 데이터가 1개이기 때문에 검은색 데이터는 노란색 클래스로 분류됩니다. 이렇게 KNN 알고리즘은 비교적 간단한 과정을 통해 이루어지는 것을 알 수 있었습니다. 그림 20은 KNN 알고리즘 수행 과정에 대한 내용입니다.

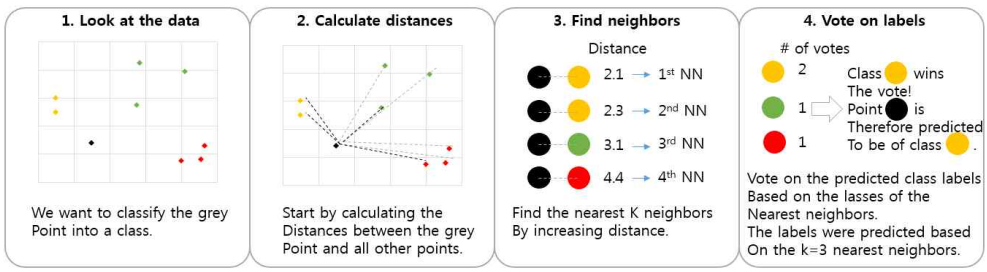


그림 20 KNN 알고리즘 수행 과정

그렇다면 이번에는 조금 더 심화하여 K 값을 정하는 방법에 대해 알아보겠습니다. 앞에서의 예제에서는 간단히 K 값을 설정했지만, KNN 알고리즘에서 K 값을 설정하는 것은 중요한 문제입니다. 이를 K 값의 최적화 문제로

부르겠습니다.

2.1.4 K 개수 최적화 문제

(1) 데이터와의 유사도를 측정 할 다른 데이터의 개수를 정하는 문제
(2) 일반적으로 K 값이 클 때 데이터인 전체적인 노이즈(Noise)를 줄일 수 있으나, 작고 중요한 패턴을 무시하는 위험 존재
(3) 과거 연구들에 의하면 대부분의 데이터 셋에서 최적의 K 값은 3~10 사이
(4) 최적의 K 값을 찾기 위해 Cross-validation 방법을 사용

- Cross validation
  - 훈련 데이터 셋(Training Data Set)과 독립적인 검증 집합(Validation Set)을 준비하여 훈련 중인 모델이 과적합(Over-fitting) 또는 과소적합(Under-fitting)인지 아닌지를 검증, 감시하는 방법

- K-fold cross validation
  - 훈련 데이터를 K 등분한 후 K-1개의 데이터 셋은 훈련 데이터로 사용하고 나머지 1개의 데이터 셋은 검증을 위해 사용한다. 여기서 검증을 위한 1개의 데이터 셋을 바꿔가며 K번 반복하여 검증하며 모델을 훈련

K 값은 분류하고자 하는 데이터와 가장 가까운 다른 데이터들의 개수로, K 값에 따라서 모델 성능과 분류 결과에 많은 차이가 생길 수 있습니다. 일반적으로 K값은 그 값이 클 때 데이터의 전체적인 노이즈를 줄일 수 있습니다. K 값이 너무 작을 때 발생할 수 있는 문제를 과적합 문제, 너무 클 때 발생할 수 있는 문제를 과소적합 문제라고 칭합니다. 그림 21은 K-fold cross-validation 과정에 대한 내용입니다.

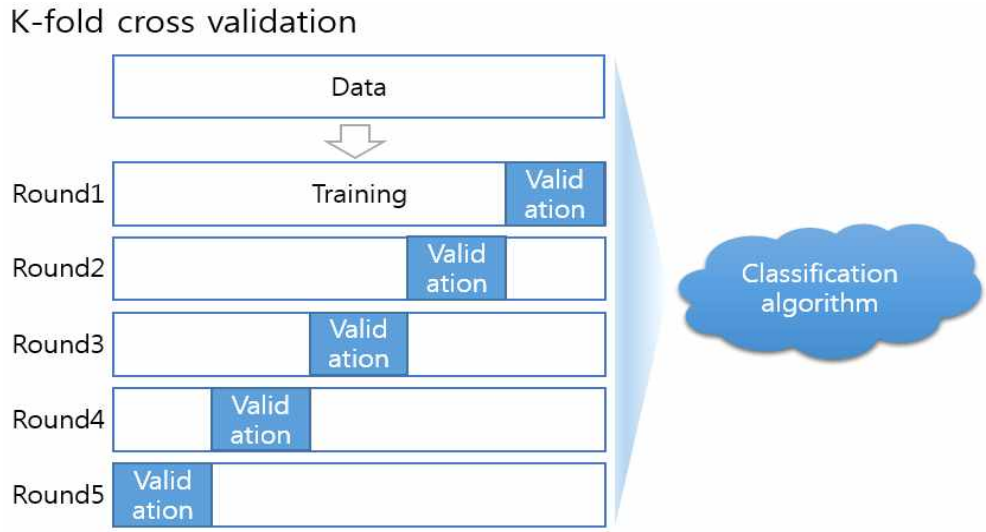


그림 21 K-fold cross validation

(5) 과적합(Over-fitting)

- K=1일 때, 즉 데이터와 가장 가까운 1개 데이터의 라벨을 새로운 데이터의 라벨로 할당하기 때문에 잘못된 분류를 야기할 수 있음

(6) 과소적합(Under-fitting)

- K의 값이 너무 클 때(ex. K=전체 데이터 개수) 새로운 데이터의 라벨은 항상 전체 데이터의 대다수를 차지하는 라벨로 분류

과적합 문제는 관측치인 K 값이 너무 작기 때문에 오분류로 이어지는 경우를 의미하며, 과소적합 문제는 K 값이 너무 크기 때문에 데이터의 대다수를 차지하는 라벨로 오분류 될 확률이 높은 경우를 의미합니다.

다음 그래프 중 1번은 과소적합한 경우, 2번은 학습이 잘 된 경우, 3번은 과적합한 경우를 나타낸 그래프입니다. 그림 22는 과적합, 과소적합과 학습이 잘된 경우의 그래프입니다.

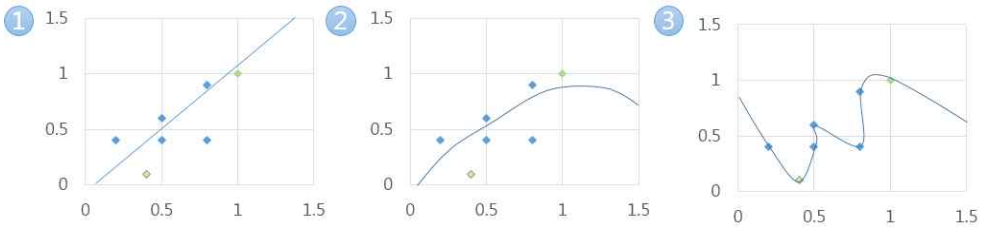


그림 22 과적합, 과소적합, 학습이 잘 된 경우의 그래프

즉, KNN 알고리즘에서 K 값은 알고리즘의 성능에 매우 중요한 역할을 합

니다. 실습을 하기 전 먼저 머신러닝을 위한 예제 데이터와 관련 함수를 제공하는 scikit-learn 라이브러리에 대해 간단히 살펴보겠습니다.

2.2 Scikit-learn(Sklearn) 패키지 소개

2.2.1 Scikit-learn 패키지

- Python에서 Machine Learning(머신러닝)을 위한 데이터 및 함수를 제공하는 대표적인 패키지
- Machine Learning의 Classification(분류), Regression(회귀), Clustering(군집) 등 다양한 알고리즘들이 제공
- 공식 사이트 : http://scikit-learn.org/stabel/

Scikit-learn 패키지는 머신러닝을 위한 예제 데이터와 관련 함수를 제공하는 라이브러리로, 분류, 회귀, 군집 등 다양한 알고리즘을 제공합니다.

2.2.2 Scikit-learn 대표 기능 및 함수 설명

(1) 예제 데이터 셋 로드 함수
- ex) sklearn.datasets.load_iris(), sklearn.datasets.load_flights() 등

(2) 학습 및 예측
- ex) sklearn.svm.SVC.fit() : 모델에 맞게 학습하는 함수
- ex) sklearn.svm.SVC.predict() : 새로운 값을 예측하는 함수, 즉 새로운 데이터의 라벨을 예측하는 함수

(3) 파라미터 재설정 및 재학습 함수
- ex) sklearn.svm.SVC.set_params() : 모델의 파라미터를 변경하는 함수
- ex) sklearn.svm.SVC.set_params.fit() : 모델의 파라미터를 변경하고 다시 모델에 맞게 학습하는 함수

scikit-learn 라이브러리는 분류, 회귀, 군집 등의 문제에 대한 다양한 알고리즘을 제공하고 있습니다. 대표적인 기능을 살펴볼까요?

먼저 scikit-learn에서 제공하는 다양한 데이터 셋을 로드할 수 있는 함수들이 존재합니다. 이는 scikit-learn의 서브패키지 datasets에서 load_iris() 함수와 같이 load로 시작하는 함수를 통해 가져올 수 있습니다. 두 번째로 모델을 학습시키고 예측하는 함수는 각각 fit()함수와 predict() 함수입니다. 그리고 파라미터를 변경하고 재학습시키기 위해 set_params() 함수와 set_params.fit() 함

수를 각각 제공하고 있습니다. 이제 scikit-learn 라이브러리를 이용해 KNN 알고리즘을 이용한 분류 실습을 시작해보겠습니다.

2.3 KNN 실습

2.3.1 예제 1 : Scikit-learn 패키지를 이용한 분류 예제 - Iris Data 분류

(1) Python 패키지 로드

먼저 사용할 python 패키지를 import해보겠습니다. 첫 번째 줄에서는 KNN 알고리즘을 수행하는 KNeighborsClassifier() 함수를 import하고, 두 번째 줄에서는 모델 성능 평가를 위한 metrics 패키지의 accuracy_score()함수를 import합니다.

그리고 세 번째 줄에서 cross validation을 위해 데이터를 나누어주는 train_test_split() 함수를 import하겠습니다. cross validation에 대해서는 뒤에서 다루도록 하겠습니다.

다섯 번째, 여섯 번째 줄에서 pandas와 numpy 라이브러리를 각각 pd, np라는 이름으로 불러오겠습니다. 그림 23은 패키지 import하는 내용입니다.

```
1 from sklearn.neighbors import KNeighborsClassifier
2 from sklearn.metrics import accuracy_score
3 from sklearn.cross_validation import train_test_split
4
5 import pandas as pd
6 import numpy as np
```

그림 23 패키지 import

(2) Iris Data 로드

이제 scikit-learn의 내장 데이터 셋인 iris 데이터를 가져오겠습니다. 두 번째 줄에서 scikit-learn의 서브패키지인 datasets의 load_iris() 함수를 사용해 가져온 데이터 셋을 iris 변수에 저장합니다. 그리고 iris 변수의 data 값을 X 변수에, target 값을 y 변수로 각각 선언합니다. 여기서 X 변수는 데이터의 feature 값들이며, y 변수는 분류된 데이터의 라벨 값을 의미합니다. 그림 24는 iris data를 로드하는 내용입니다.

```
# import data set
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

그림 24 Iris data 로드

(3) Iris Data 데이터프레임으로 확인

데이터를 출력하여 확인해볼까요? 데이터를 테이블 형태로 쉽게 보기 위하여 데이터프레임 형태로 만들어 보았습니다. 먼저 첫 번째 줄에서 pandas 라이브러리의 DataFrame() 함수를 이용해 데이터프레임을 생성합니다. 다섯 번째 줄에서는 데이터의 라벨 값들을 확인하기 위하여 numpy의 unique() 함수를 사용하여 중복이 제거된 y 변수 값을 출력하였습니다.

그리고 print()문 안에서 출력되는 세 값은 numeric 값이기 때문에 앞의 문자열과 함께 출력하기 위해 str()함수를 사용하여 str 형으로 변환하여 사용합니다. 출력된 결과를 확인해보겠습니다. iris data의 총 개수는 150개이며, sepal length, sepal width, petal length, petal width 총 네 개의 feature 값들을 가지고 있습니다. 그리고 0, 1, 2 세 개의 라벨로 분류된 데이터 셋임을 확인할 수 있습니다. 그림 25는 iris data를 데이터프레임 형태로 출력 및 확인하는 내용입니다.

```
df = pd.DataFrame(X, columns = iris.feature_names)
print("< Iris Data >")
print("The number of sample data : " + str(len(df)))
print("The number of features of the data : " + str(len(df.columns)))
print("The labels of the data : " + str(np.unique(y)))
df
```

```
< Iris Data >
The number of sample data : 150
The number of features of the data : 4
The labels of the data : [0 1 2]
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4

그림 25 Iris data 데이터프레임 형식으로 출력 및 확인

(4) Training, Test data set 분리

이제 전체 데이터를 training data set, test data set으로 나누고 training data로 모델을 생성하여 test data의 라벨을 예측하여 모델의 성능을 측정해보겠습니다. 전체 데이터를 훈련 데이터와 시험 데이터로 나누기 위해서는 앞에서 impot한 train_test_split() 함수를 사용합니다.

train_test_split() 함수의 파라미터에 순서대로 X, y값을 넣고 test_size 파라미터를 사용해 전체 데이터에서 test data로 사용할 비율을 33%로 설정합니다. random_state 파라미터는 데이터를 샘플링할 때 사용하는 시드 값으로, 이 값을 같은 숫자로 설정하면 나중에도 동일하게 데이터를 샘플링할 수 있습니다. 시드 값은 실험 재현(reproduction) 측면에서 중요하게 사용되기 때문에, 다음에도 동일한 실험 결과를 낳을 수 있게 하기 위해서 가급적이면 명시하는 것이 좋습니다.

이렇게 호출한 train_test_split() 함수는 X 값을 분리한 값, y 값을 분리한 값을 반환합니다. 즉, X_train 변수에 training data X 값이 저장되고, X_test 변수에 test data X 값이 저장됩니다. y_train 변수에는 training data y 값이,

y_test 변수에는 test data y 값이 저장되겠죠.

여섯 번째 줄과 일곱 번째 줄에서 X_train 변수와 X_test 변수의 개수를 출력한 결과를 살펴보면, 위의 함수에서 설정한 것처럼 test data가 전체 데이터 150개 중 약 33%인 50개가 샘플링 되었고, 그 나머지가 training data인 것을 확인할 수 있습니다. 그림 26은 전체 데이터를 training data와 test data로 분리하는 내용입니다.

```
# split whole data set into train set and test set
# test_size : the proportion of the dataset to include in the test split. (0~1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33,
                                                    random_state = 42)

print("The number of train data set : %d " %len(X_train))
print("The number of test data set : %d " %len(X_test))
```

```
The number of train data set : 100
The number of test data set : 50
```

그림 26 training data와 test data로 분리

(6) 임의의 K 설정, 모델 학습 및 정확도 측정

KNeighborClassifier() 함수를 호출하여 기본 KNN 모델을 생성하여 estimator 변수로 선언합니다. 여기서 K 값을 임의로 3으로 설정하겠습니다. 이는 함수의 n_neighbors 파라미터로 설정할 수 있습니다. 그리고 네 번째 줄에서 훈련 데이터 X 값과 y 값을 넣은 X_train, y_train 변수를 넣어 모델을 학습시킵니다.

그리고 여섯 번째 줄에서 test data X 값인 변수 X_test를 넣어 반환되는 결과 y 값을 label_predict 변수로 선언하겠습니다. 즉, label_predict 변수에는 training data로 학습시킨 모델로 분류한 test data의 라벨 값이 저장됩니다.

이제 결과가 나왔으니 분류 모델의 정확도를 측정하는 accuracy_score() 함수를 사용하여 모델의 성능을 평가해보겠습니다. accuracy_score()함수는 실제 라벨 값과 예측한 라벨 값을 순서대로 파라미터로 사용합니다. 즉, y_test 변수와 label_predict 변수를 넣어 accuracy_score() 함수를 호출합니다. 그리고 이 값을 소수점 9번째 자리까지 출력하기 위하여 % 연산자로 문자열 형변환을 하여 출력하였습니다. 결과적으로 이 모델의 정확도는 98%로 상당히 높은

값이 나온 것을 확인할 수 있습니다. 그림 27은 임의의 K 값 설정, 모델 학습 및 정확도 측정에 대한 내용입니다.

```
# instantiate learning model (k = 3)
estimator = KNeighborsClassifier(n_neighbors=3)
# fitting the model
estimator.fit(X_train, y_train)
# predict the response
label_predict = estimator.predict(X_test)
# evaluate accuracy
print("The accuracy score of classification: %.9f"
      %accuracy_score(y_test, label_predict))
```

```
The accuracy score of classification: 0.980000000
```

그림 27 임의의 K 값 설정, 모델 학습 및 정확도 측정

이번에는 최적의 K 값을 찾는 실습을 해보겠습니다. 앞의 이론에서 설명한 것처럼 K 값에 따라 분류 결과가 달라질 수 있기 때문에, K 값의 최적화는 중요한 문제입니다. 먼저 K 값으로 사용할 후보 값들을 만들어보겠습니다. 그리고 이번에는 모델의 성능을 평가하는 방법인 cross-validation 방법을 이용하여 K 값에 따른 정확도를 구해보겠습니다.

먼저 cross validation 방법에 대해 살펴보겠습니다. cross-validation이란 전체 데이터를 training data와 test data로 교차적으로 나누어 성능을 검증하는 방법입니다. 앞에서 간단히 다뤘던 것처럼 모델은 training data에 과적합되어 새로운 데이터를 잘 예측할 수 없을 수 있기 때문에 이러한 과적합 문제를 피하기 위해서 많이 사용하는 방법입니다. 그 중 대표적인 방법 중 하나인 K-fold cross validation을 예제로 설명해보겠습니다. 방법은 다음과 같습니다. 전체 데이터를 K등분 한 후 K-1개의 데이터는 training data로 나머지 1개의 데이터는 test data로 샘플링합니다.

그 다음 training data로 모델을 학습시킨 후 test data를 넣어 모델의 성능을 평가합니다. 그림처럼 한 번의 라운드를 거친 후 training data를 바꿔가며 모델의 성능을 평가합니다. 최종적으로 각 라운드에서 구한 K 개의 성능 평가 값을 산술 평균하여 모델을 평가할 수 있습니다. 우리는 이 방법을 사용해

K 값에 따른 성능 평가를 해보겠습니다.

(7) 최적의 K 값을 찾기 위한 cross validation 방법 사용하기

먼저 네 번째 줄에서 1부터 50 사이의 연속적인 값을 리스트 형태로 변수 myList에 저장합니다. 그리고 다섯 번째 줄에서는 single line for loops를 사용하였는데, 이는 for 반복문을 한 줄로 요약하여 표현하는 것을 의미합니다. 이를 간단히 설명해보면, 변수 myList에 반복자를 생성하여 변수 안의 각 요소를 if 문으로 홀수 일 때 추출하여 리스트 안에 담는 것입니다. 즉, 1부터 50까지의 수 중 홀수만 추출하여 리스트 형태로 만들어 변수 neighbors로 선언한 것입니다.

여섯 번째 줄에서 neighbors 변수를 출력하여 일곱 번째 줄에서 그 개수를 출력해 보았습니다. 역시나 1부터 49까지 25개의 홀수만 출력된 것을 확인할 수 있습니다. K 값의 후보들이 만들어졌으니 cross-validation을 하여 K 값에 따른 성능 평가를 해보겠습니다. 그림 28은 K 값으로 사용할 후보 값 생성에 대한 내용입니다.

```
1 # perform 10-fold cross validation
2
3 # create odd list of k for kNN
4 myList = list(range(1,50))
5 neighbors = [ x for x in myList if x % 2 != 0]
6 print(neighbors)
7 print("The number of neighbors k is %d" %len(neighbors))
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 4
7, 49]
The number of neighbors k is 25
```

그림 28 K 값으로 사용할 후보 값 생성

(8) 최적의 K 값을 찾기 위한 cross validation 방법 사용하기

그리고 이번에는 모델의 성능을 평가하는 방법인 cross-validation 방법을 이용 하여 K 값에 따른 정확도를 구해보겠습니다. 네 번째 줄에서는 K의 후보 값인 변수 neighbors에 반복자를 생성하여 for 반복문을 수행합니다. 다섯 번째 줄에서 K 값을 출력합니다. 그리고 여섯 번째 줄에서 K 값을 넣은 모델을 생성한 후 일곱 번째 줄에서 cross_val_score() 함수를 호출합니다. cross validation을 수행하는 cross_val_score()함수는 input 값으로 모델과 학습할 훈

련 데이터를 갖습니다. 즉, estimator, X_train, y_train 변수를 넣고 cv 값에 10을 넣어 10-fold cross validation을 수행하였습니다. 그리고 scoring 파라미터에 accuracy 문자열을 넣어 cross validation의 정확도 값을 구할 것을 설정합니다. 함수의 output 값으로 나오는 10개의 정확도 값은 변수 scores에 저장합니다. 그리고 9번째 줄에서 10개의 정확도 값을 출력하였습니다. 그리고 이 값들의 평균값을 mean()함수로 계산하여 리스트 변수 cv_scores에 추가합니다. 즉, 변수 cv_scores에는 K 값에 따른 cross validation 평균 정확도 값들이 리스트 형태로 저장됩니다. 출력된 결과를 확인해볼까요?

K 값에 따라 아래에 cross validation을 통해 구해진 10개의 정확도 값이 출력되었고, 그 아래에 평균 정확도 값이 출력된 것을 확인할 수 있습니다. 지금까지 25개의 K 값에 따른 모델의 성능평가를 하였습니다. 그림 29는 K 값에 따른 성능평가에 대한 내용입니다.

```
1 # empty list that will hold cross validation scores
2 cv_scores = []
3 # perform 10-fold cross validation
4 for k in neighbors:
5     print("< k = %d >" %k)
6     estimator = KNeighborsClassifier(n_neighbors=k)
7     scores = cross_val_score(estimator, X_train, y_train, cv = 10, scoring = 'ac
8     print("The scores of classification are \n" + str(scores))
9     cv_scores.append(scores.mean()) # average error
10    print("The average score of scores is %.9f \n" %scores.mean())
```

```
< k = 1 >
The scores of classification are
[ 1.         0.90909091 1.         0.72727273 0.9        1.         1.
  1.         1.         0.88888889]
The average score of scores is 0.942525253

< k = 3 >
The scores of classification are
[ 0.91666667 1.         1.         0.72727273 0.9        1.         1.
  1.         1.         0.88888889]
The average score of scores is 0.943282828

< k = 5 >
The scores of classification are
[ 1.         1.         1.         0.72727273 0.9        1.         1.
  1.         1.         0.88888889]
The average score of scores is 0.951616162
```

그림 29 K 값에 따른 모델 성능평가

(9) 최적의 K 값 찾기 위한 cross validation 방법 사용하기

K 값에 따른 모델의 성능을 비교하기 위하여 MSE, 즉 misclassification rate 를 사용할건데요. 이는 오분류 비율이라고 해석할 수 있습니다. 오분류 비율은 모델의 분류가 잘못된 비율을 의미하는 것으로 값이 높을수록 안 좋은 모델이라고 할 수 있습니다.

이 오분류 비율은 먼저 구한 정확도 값을 변환하여 만들어보겠습니다. 세 번째 줄을 보면 오분류 값은 1 에서 정확도 값 cv_scores를 뺀 값으로 MSE 변수로 선언합니다. 그리고 K 값에 따른 오분류 값을 쉽게 그래프로 보기 위해 pyplot 패키지를 이용하여 plot으로 나타내어 보았습니다. 또한 열두 번째 줄에서는 오분류 비율이 가장 작은 K 값을 찾았습니다.

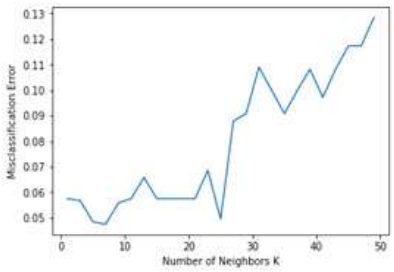
또한 12, 13, 14번째 줄에서는 오분류 비율이 가장 작은 K 값을 찾았습니다. 먼저 열두 번째 줄에서는 오분류 비율 값이 저장된 MSE 변수의 최솟값을 min() 함수로 구해 변수 min_MSE로 선언하였습니다. 즉, 변수 min_MSE 에는 최소 오분류 비율 값이 저장되었습니다. 그리고 열세 번째 줄에서는 변수 MSE 안에서 변수 min_MSE의 인덱스 번호를 찾기 위해 index()함수를 사용하여 이를 변수 index_of_min_MSE로 선언하였습니다.

그리고 마지막으로 K 후보 값들이 저장된 neighbors 변수에 인덱싱 함수인 대괄호 안에 이 인덱스 번호를 넣어, 최소 오분류 비율 값을 가지는 K 값을 변수 optimal_k로 선언하였습니다. 즉, 최적의 k 값이 저장된 것입니다. 출력된 결과를 보면 최적의 K 값은 7로 가장 낮은 오분류 값을 보임을 그래프로 확인할 수 있습니다. 그림 30은 K 값에 따른 오분류 비율에 대한 내용입니다.

```
# changing to misclassification rate (a.k.a classification error)
# MSE = 1 - cross validation score
MSE = [1 - x for x in cv_scores]

# plot misclassification error vs k
plt.plot(neighbors, MSE)
plt.xlabel("Number of Neighbors K")
plt.ylabel("Misclassification Error")
plt.show()

# determining best k
min_MSE = min(MSE)
index_of_min_MSE = MSE.index(min_MSE)
optimal_k = neighbors[index_of_min_MSE]
print ("The optimal number of neighbors i is %d" % optimal_k)
```




```
The optimal number of neighbors i is 7
```

그림 30 K 값에 따른 오분류 비율

이번에는 새로운 데이터를 KNN 알고리즘으로 분류하는 실습을 해보겠습니다. 이번에 사용할 데이터는 역시 scikit-learn의 내장 데이터 셋 중 하나인 breast_cancer 데이터입니다.

2.3.2 예제 2 : Scikit-learn 패키지를 이용한 분류 예제 - Breast Cancer Data 분류

(1) Python 패키지 가져오기 및 데이터 로드 및 출력

먼저 첫 번째 줄에서 load_breast_cancer() 함수를 import하겠습니다. 그리고 세 번째 줄에서 로드한 데이터 셋을 breast_cancer 변수로 선언하고 data값, target값을 각각 X변수, y변수로 선언합니다. 마찬가지로 데이터프레임으로 만들어 확인해보겠습니다. 출력된 결과처럼 breast cancer 데이터는 총 569개의 데이터로 30개의 feature 값들을 가지고 있으며 0,1 두 개의 라벨로 분류되었음을 확인할 수 있습니다. 그림 31은 패키지 import, 데이터 로드 및 출력에 대한 내용입니다.

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import normalize

breast_cancer = load_breast_cancer()
X = breast_cancer.data
y = breast_cancer.target

df = pd.DataFrame(X, columns = breast_cancer.feature_names)
df
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	me sy
0	17.990	10.38	122.80	1001.0	0.11840	0.27760	0.300100	0.147100	0.2
1	20.570	17.77	132.90	1326.0	0.08474	0.07864	0.086900	0.070170	0.1
2	19.690	21.25	130.00	1203.0	0.10960	0.15990	0.197400	0.127900	0.2
3	11.420	20.38	77.58	386.1	0.14250	0.28390	0.241400	0.105200	0.2
4	20.290	14.34	135.10	1297.0	0.10030	0.13280	0.198000	0.104300	0.1
5	12.450	15.70	82.57	477.1	0.12780	0.17000	0.157800	0.080890	0.2
6	18.250	19.98	119.60	1040.0	0.09463	0.10900	0.112700	0.074000	0.1
7	13.710	20.83	90.20	577.9	0.11890	0.16450	0.093660	0.059850	0.2
8	13.000	21.82	87.50	519.8	0.12730	0.19320	0.185900	0.093530	0.2
9	12.460	24.04	83.97	475.9	0.11860	0.23960	0.227300	0.085430	0.2

그림 31 패키지 import, 데이터 로드 및 출력

(2) Training, Test data 분리

이제 전체 데이터를 training data set, test data set 으로 나누어보겠습니다. 마찬가지로 train_test_split() 함수를 사용하여 데이터를 나눈 후 그 개수를 출력해보았습니다. 전체 569개의 데이터 중 약 33%인 188개의 데이터가 test data로 할당된 것을 확인할 수 있습니다. 그림 32는 전체 데이터를 training data와 test data로 분리하는 내용입니다.

```
# split whole data set into train set and test set
# test_size : the proportion of the dataset to include in the test split. (0~1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33,
                                                    random_state = 42)

print("The number of train data set : %d " %len(X_train))
print("The number of test data set : %d " %len(X_test))
```

```
The number of train data set : 381
The number of test data set : 188
```

그림 32 전체 데이터를 training data와 test data로 분리

(3) 모델 학습 및 정확도 측정하기

이번에는 K 값을 임의로 5로 설정하였습니다. 그리고 이번에는 weights 파라미터를 추가하여 KNeighborClassifier() 함수를 호출하였습니다. 이론 강의에서 분류할 데이터와 다른 데이터들과의 거리를 측정한 후 가까운 K개의 데이터를 거리 순으로 랭킹을 매겨 이웃 데이터로 지정했던 것을 기억하시나요? weights 파라미터는 이와 관련이 있는데요. weights는 K개의 데이터에 가중치를 주는 방식을 설정하는 파라미터로, distance로 설정하면 거리에 따라 다른 가중치를 준 후 분류에 영향을 주게 됩니다.

즉, 더 가까운 데이터가 분류에 더 많은 영향을 주게 됩니다. 이렇게 생성한 KNN 모델을 estimator 변수로 선언하고 네 번째 줄에서 training data를 넣어 모델을 학습시키겠습니다. 그리고 여섯 번째 줄에서 X_test 변수, 즉 test data의 X 값을 넣어 예측한 라벨 값을 label_predict 변수로 선언하였습니다. 그리고 여덟 번 째 줄에서 분류 정확도 값을 출력해보았습니다. 약 94%의 정확도를 나타내는 것을 확인할 수 있었습니다. 그림 33은 모델 학습 및 정확도 측정에 대한 내용입니다.

```
1 # instantiate learning model (k = 3)
2 estimator = KNeighborsClassifier(n_neighbors=5, weights = 'distance')
3 # fitting the model
4 estimator.fit(X_train, y_train)
5 # predict the response
6 label_predict = estimator.predict(X_test)
7 # evaluate accuracy
8 print("The accuracy score of classification: %.9f"
9       %accuracy_score(y_test, label_predict))
```

```
The accuracy score of classification: 0.941489362
```

그림 33 모델 학습 및 정확도 측정

이번에는 최적의 K 값을 찾는 실습을 해보겠습니다. 이론에서 설명한 것처럼 K 값에 따라 분류 결과가 달라질 수 있기 때문에, K 값의 최적화는 중요한 문제입니다. 먼저 K 값으로 사용할 후보 값들을 만들어보겠습니다. 그리고 이번에는 모델의 성능을 평가하는 방법인 cross-validation 방법을 이용하여 K 값에 따른 정확도를 구해보겠습니다.

먼저 cross validation 방법에 대해 살펴보겠습니다. cross-validation이란 전체

데이터를 training data와 test data로 교차적으로 나누어 성능을 검증하는 방법입니다. 앞에서 간단히 다뤘던 것처럼 모델은 훈련 데이터에 과적합되어 새로운 데이터를 잘 예측할 수 없을 수 있기 때문에 이러한 과적합 문제를 피하기 위해서 많이 사용하는 방법입니다.

그 중 대표적인 방법 중 하나인 K-fold cross validation을 예제로 설명해보겠습니다. 방법은 다음과 같습니다. 전체 데이터를 K등분 한 후 K-1개의 데이터는 훈련 데이터로 나머지 1개의 데이터는 검증 데이터로 샘플링합니다.

그 다음 훈련 데이터로 모델을 학습시킨 후 test data를 넣어 모델의 성능을 평가합니다. 그림처럼 한 번의 라운드를 거친 후 훈련 데이터를 바꿔가며 모델의 성능을 평가합니다. 최종적으로 각 라운드에서 구한 K 개의 성능 평가 값을 산술 평균하여 모델을 평가할 수 있습니다. 우리는 이 방법을 사용해 K 값에 따른 성능 평가를 해보겠습니다.

(4) 최적의 K 값을 찾기 위한 cross validation 방법 사용하기

먼저 네 번째 줄에서 1부터 100 사이의 연속적인 값을 리스트 형태로 변수 myList에 저장합니다. 그리고 다섯 번째 줄에서는 single line for loops를 사용하였는데, 이는 for 반복문을 한 줄로 요약하여 표현하는 것을 의미합니다. 이를 간단히 설명해보면, myList 변수에 반복자를 생성하여 변수 안의 각 요소를 if 문으로 홀수 일 때 추출하여 리스트 안에 담는 것입니다.

즉, 1부터 100까지의 수 중 홀수만 추출하여 리스트 형태로 만들어 변수 neighbors로 선언한 것입니다. 여섯 번째 줄에서 변수 neighbors를 출력하여 일곱 번째 줄에서 그 개수를 출력해 보았습니다. 역시나 1부터 99까지 50개의 홀수만 출력된 것을 확인할 수 있습니다. 그림 34는 K 값으로 사용할 후보 값 생성에 대한 내용입니다.

```
# perform 10-fold cross validation

# create odd list of k for kNN
myList = list(range(1,100))
neighbors = [ x for x in myList if x % 2 != 0]
print(neighbors)
print("The number of neighbors k is %d" %len(neighbors))
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 4
7, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 9
1, 93, 95, 97, 99]
The number of neighbors k is 50
```

그림 34 K 값으로 사용할 후보 값 생성

(5) 최적의 K 값을 찾기 위한 cross validation 방법 사용하기

K 값의 후보들이 만들어졌으니 cross-validation을 하여 K 값에 따른 성능 평가를 해보겠습니다. 4번째 줄에서는 K 후보인 변수 neighbors에 반복자를 생성하여 for 반복문을 수행합니다.

먼저 다섯 번째 줄에서 K 값을 출력합니다. 그리고 여섯 번째 줄에서 K 값을 넣은 모델을 생성한 후 일곱 번째 줄에서 cross_val_score() 함수를 호출합니다.cross validation을 수행하는 cross_val_score()함수는 input 값으로 모델과 학습할 훈련 데이터를 갖습니다. 즉, estimator, X_train, y_train 변수를 넣고 cv 값에 10을 넣어 10-fold cross validation을 수행하였습니다.

그리고 scoring 파라미터에 accuracy 문자열을 넣어 cross validation의 정확도 값을 구할 것을 설정합니다. 함수의 output 값으로 나오는 10개의 정확도 값은 scores 변수에 저장합니다. 그리고 아홉 번째 줄에서 10개의 정확도 값을 출력하였습니다.

그리고 이 값들의 평균값을 mean()함수로 계산하여 cv_scores 리스트 변수에 추가합니다. 즉, cv_scores 변수에는 K 값에 따른 cross validation 평균 정확도 값들이 리스트 형태로 저장됩니다. 출력된 결과를 확인해볼까요? K 값에 따라 아래에 cross validation을 통해 구해진 10개의 정확도 값이 출력되었고, 그 아래에 평균 정확도 값이 출력된 것을 확인할 수 있습니다. 지금까지 50개의 k값에 따른 모델의 성능 평가를 하였습니다. 그림 35는 K 값에 따른 성능평가에 대한 내용입니다.

```
# perform 10-fold cross validation
for k in neighbors:
    print("< k = %d >" %k)
    estimator = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(estimator, X_train, y_train, cv = 10, scoring = 'accura
    print("The scores of classification are ₩n" + str(scores))
    cv_scores.append(scores.mean()) # average error
    print("The average score of scores is %.9f ₩n" %scores.mean())
```

```
< k = 1 >
The scores of classification are
[ 0.97435897  0.84615385  0.92307692  0.97435897  0.82051282  0.89473684
  0.86486486  0.91891892  0.89189189  0.83783784]
The average score of scores is 0.894671189

< k = 3 >
The scores of classification are
[ 0.94871795  0.87179487  0.8974359   0.94871795  0.94871795  0.89473684
  0.86486486  0.89189189  0.94594595  0.81081081]
The average score of scores is 0.902363497

< k = 5 >
The scores of classification are
[ 0.94871795  0.87179487  0.8974359   0.94871795  0.94871795  0.92105263
  0.91891892  0.89189189  0.91891892  0.81081081]
The average score of scores is 0.907697779

< k = 7 >
```

그림 35 K 값에 따른 성능평가

(6) 최적의 K 값을 찾기 위한 cross validation 방법 사용하기

K 값에 따른 모델의 성능을 비교하기 위하여 MSE, 즉 misclassification rate 를 사용할건데요. 이는 오분류 비율이라고 해석할 수 있습니다. 오분류 비율은 모델의 분류가 잘못된 비율을 의미하는 것으로 값이 높을수록 안 좋은 모델이라고 할 수 있습니다.

이 오분류 비율은 우리가 위에서 구한 정확도 값을 변환하여 만들어보겠습니다. 세 번째 줄을 보면 오분류 값은 1 에서 정확도 값 cv_scores를 뺀 값으로 변수 MSE로 선언합니다. 그리고 K 값에 따른 오분류 값을 쉽게 그래프로 보기 위해 pyplot 패키지를 이용하여 plot으로 나타내어 보았습니다.

또한 12, 13, 14번째 줄에서는 오분류 비율이 가장 작은 K 값을 찾았습니다. 먼저 열두 번째 줄에서는 오분류 비율 값이 저장된 MSE 변수의 최솟값을 min() 함수로 구해 min_MSE 변수로 선언하였습니다. 즉, min_MSE 변수에는 최소 오분류 비율 값이 저장되었습니다.

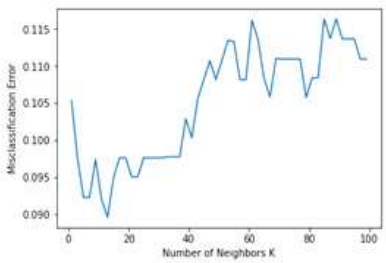
그리고 열세 번째 줄에서는 MSE 변수 안에서 min_MSE 변수의 인덱스 번호를 찾기 위해 index()함수를 사용하여 이를 index_of_min_MSE 변수로 선언하였습니다.

그리고 마지막으로 K 후보 값들이 저장된 neighbors 변수에 인덱싱 함수인 대괄호 안에 이 인덱스 번호를 넣어, 최소 오분류 비율 값을 가지는 k 값을 optimal_k 변수로 선언하였습니다. 즉, 최적의 K 값이 저장된 것입니다. 출력된 결과를 보면 최적의 K값은 13으로 가장 낮은 오분류 값을 보임을 그래프로 확인할 수 있습니다. 그림 36은 K 값에 따른 오분류 비율에 대한 내용입니다.

```
# changing to misclassification rate (a.k.a classification error)
# MSE = 1 - cross validation score
MSE = [1 - x for x in cv_scores]

# plot misclassification error vs k
plt.plot(neighbors, MSE)
plt.xlabel("Number of Neighbors K")
plt.ylabel("Misclassification Error")
plt.show()

# determining best k
min_MSE = min(MSE)
index_of_min_MSE = MSE.index(min_MSE)
optimal_k = neighbors[index_of_min_MSE]
print ("The optimal number of neighbors i is %d" % optimal_k)
```




```
The optimal number of neighbors i is 13
```

그림 36 K 값에 따른 오분류 비율

(7) 새로운 K 값으로 모델 학습 및 정확도 측정하기

이제 앞에서 찾은 최적의 K 값으로 다시 모델을 생성하여 모델 정확도를 구해보았습니다. 결과처럼 기존에 임의의 수 5로 정한 K값을 13으로 변경한 후에 정확도가 약 94%에서 약 96%로 성능이 높아진 것을 확인할 수 있습니

다. 그림 37은 새로운 K 값으로 모델 학습 및 정확도 측정에 대한 내용입니다.

```
# instantiate learning model (k = 3)
estimator = KNeighborsClassifier(n_neighbors=13)
# fitting the model
estimator.fit(X_train, y_train)
# predict the response
label_predict = estimator.predict(X_test)
# evaluate accuracy
print("The accuracy score of classification: %.9f"
      %accuracy_score(y_test, label_predict))
```

```
The accuracy score of classification: 0.962765957
```

그림 37 새로운 K 값으로 모델 학습 및 정확도 측정

이렇게 cross validation 방법과 최적의 K 값을 찾는 실습까지 KNN 알고리즘에 대해 알아보았습니다. KNN은 간단하지만 꽤 높은 정확도를 내는 것을 실습을 통해 확인할 수 있었습니다.