Ch.4 Clustering

1. 지도, 비지도학습

일반적으로 기계학습에서 학습은 크게 **지도학습과 비지도학습**으로 나뉜다.
지도학습은 Label이 있는 상태에서의 학습을 이야기하는데, 보통 분류와 회귀가 있다.
한편, 비지도학습은 Label이 없는 상태에서의 학습으로 군집이나, 차원축소 등이 있다.

2-1. k-means clustering

k-means 클러스터링은 1967년 MacQueen으로부터 만들어진 알고리즘이다.
이는 거리를 기반으로 가까이에 있는 k개 만큼의 군집을 만드는 알고리즘이라고 할 수 있다.
따라서 최적의 k를 찾는 것도 이 알고리즘을 사용하는데 있어 중요한 이슈 중 하나이다.

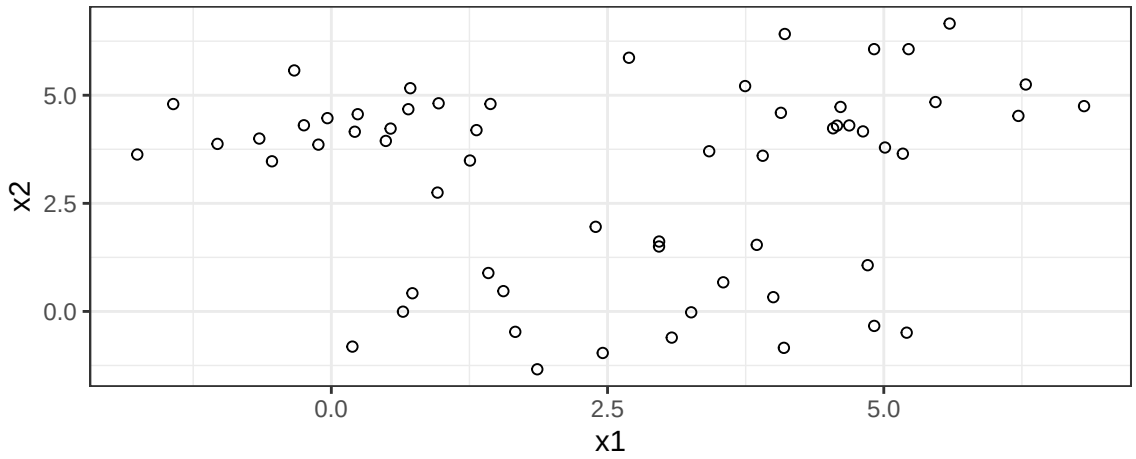
2-2. k-means clustering - procedure

k-means 알고리즘이 작동하는 방식은 다음과 같다.

1. Initialize the center of the clusters (centroid of clusters).
   - k개 군집의 중심점을 (랜덤하게) 결정해준다.
2. Attribute the closest cluster to each data point.
   - 모든 데이터에 대해 거리를 구해서, 가까운 중심점에 matching 하여 군집을 결정한다.
3. Set the position of each cluster to the mean of all data points belonging to that cluster.
   - 각 군집에 소속된 데이터들의 평균을 계산하여 중심점을 조정해준다.
4. Repeat steps 2-3 until convergence.
   - 2번과 3번 과정을 계속해서 반복하며, 최종 위치를 결정한다.
   - 더 이상 centroid가 변하지 않거나, data point에 변동이 없을 때까지 반복한다.

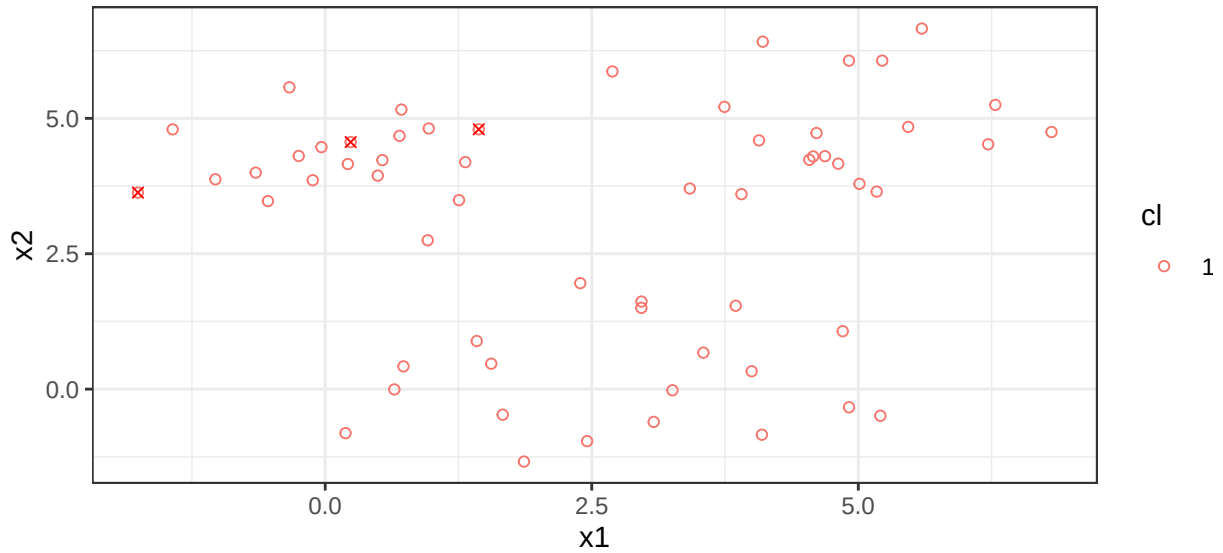
2-3. k-means clustering - code

```
# set.seed(2018)
synth.data <- data.frame(x1 = c(rnorm(20, 3, 1.5), rnorm(20, 0, 1), rnorm(20, 5, 1)),
                         x2 = c(rnorm(20, 0, 1), rnorm(20, 4, 1), rnorm(20, 5, 1)))
ndata <- nrow(synth.data)
ndim <- ncol(synth.data)

synth.data %>%
  ggplot(aes(x = x1, y = x2)) +
  geom_point(shape = 1) + theme_bw()
```



rnorm(데이터 개수, 평균, 표준편차) 함수를 통해서 정규분포를 따르는 임의의 데이터를 생성해준다. 그리고 총 60개 점에 대해서 그래프를 그리면 위와 같음을 확인할 수 있다.

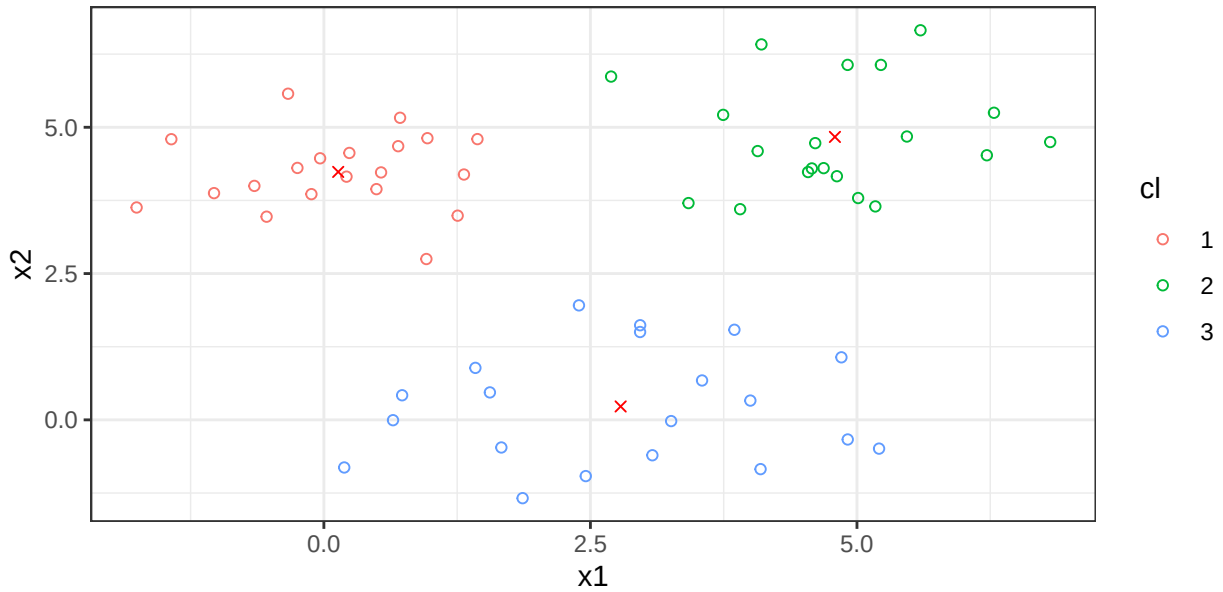
```
# Euclidean distance
u_dist <- function(x1, y1, x2, y2){
  sqrt(((x2-x1)**2) + ((y2-y1)**2))
}

# Initial Setting
k <- 3
cents <- data.frame(cl = 1:k)
cents <- cbind(cents, synth.data[sample(1:60, k), ]) # 임의로 k개 뽑아서 centroid 할당

synth.data$cl <- factor(rep(1, ndata), levels = 1:k) # 처음에는 모두 1번 군집으로 할당
synth.data %>%
  ggplot(aes(x = x1, y = x2, col = cl)) +
  geom_point(shape = 1) + theme_bw() +
  geom_point(data = cents, shape = 4, col = 'red')
```



random으로 k개의 centroid를 뽑고, 각 데이터와 중심점(빨간색 X점)을 찍어보도록 한다.

```
##### K-means 알고리즘 구현 (핵심 부분)

while (TRUE){
  past <- mean(cents$x1) + mean(cents$x2)

  for (row in 1:ndata){
    ### 1. k개의 각 군집들과의 거리를 구해주기
    for (k_value in 1:k){
      col_name <- paste0('dist_', k_value)
      if (row == 1){synth.data[, col_name] <- 0} # 거리 계산 변수 만들기

      ### 2. 모든 data point에 대해 각 군집들과 거리를 구하기
      c_df <- cents %>% filter(cl == k_value)
      synth.data[row, col_name] <- u_dist(synth.data$x1[row], synth.data$x2[row],
                                          c_df$x1, c_df$x2)}

    ### 3. 군집들과의 거리가 가장 짧은 곳으로 군집 재배치
    df <- synth.data %>% select(starts_with('dist_'))
    synth.data$cl[row] <- which.min(df[row, ])}

  ### 4. 새롭게 배치된 군집들의 평균으로 군집들의 중심점 이동
  cents <- data.frame(cl = 1:k,
                      x1 = aggregate(x1~cl, synth.data, mean)$x1,
                      x2 = aggregate(x2~cl, synth.data, mean)$x2)

  ### 5. centroid가 더 이상 변화하지 않는다면, STOP.
  new <- mean(cents$x1) + mean(cents$x2)
  if (new == past){break}
}
```

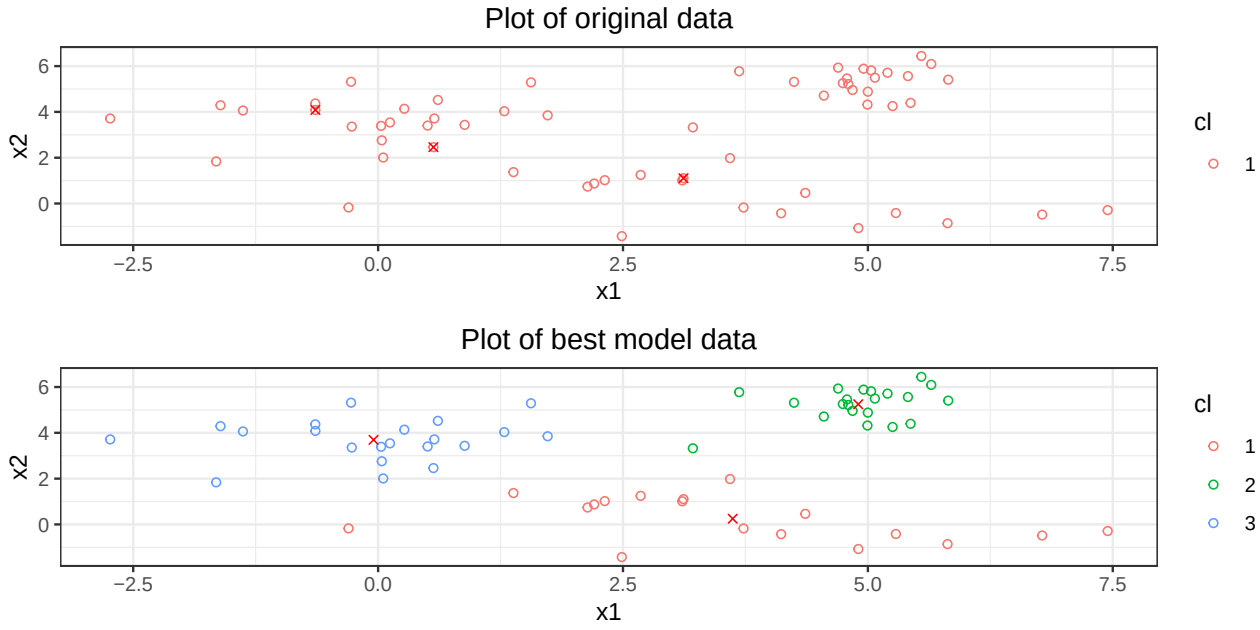
```
synth.data %>%
  ggplot(aes(x = x1, y = x2, col = cl)) +
  geom_point(shape = 1) + theme_bw() +
  geom_point(data = cents, shape = 4, col = 'red')
```



```r
##### K-means 알고리즘 구현 (전체)

k_means_algo <- function(df, x_col, y_col, k){
  ### Euclidean distance
  u_dist <- function(x1, y1, x2, y2){sqrt(((x2-x1)**2) + ((y2-y1)**2))}

  ### Initial setting & Append original data
  ndata <- nrow(df)
  cents <- data.frame(cl = 1:k)
  cents <- cbind(cents, df[sample(1:ndata, k), ])
  df[, 'cl'] <- factor(rep(1, ndata), levels = 1:k)

  output_list <- list()
  output_list['df'] <- list(df)
  output_list['cents'] <- list(cents)

  ### Algorithm
  start <- 1
  x <- deparse(substitute(x_col))
  y <- deparse(substitute(y_col))
  while (TRUE){
    past <- mean(cents$x1) + mean(cents$x2)
    for (row in 1:ndata){
      ## 1. k개의 각 군집들과의 거리를 구해주기
      for (k_value in 1:k){
        col_name <- paste0('dist_', k_value)
        if (row == 1){df[, col_name] <- 0} # 거리 계산 변수 만들기
        ## 2. 모든 data point에 대해 각 군집들과 거리를 구하기
        c_df <- cents %>% filter(cl == k_value)
        df[row, col_name] <- u_dist(df[row, x], df[row, y], c_df[, x], c_df[, y])}
      ## 3. 군집들과의 거리가 가장 짧은 곳으로 군집 재배치
      target_df <- df %>% select(starts_with('dist_'))
      df$cl[row] <- which.min(target_df[row, ])}
    ## 4. 새롭게 배치된 군집들의 평균으로 군집들의 중심점 이동
    cents <- df %>% group_by(cl) %>% summarise(x1 = mean(x1), x2 = mean(x2))
    ## 5. centroid가 더 이상 변화하지 않는다면, STOP.
    new <- mean(cents$x1) + mean(cents$x2)
    if (new == past){break}
    ## 6. 변화하는 데이터프레임 저장
    name1 <- paste0('df', start)
    name2 <- paste0('cents', start)
    output_list[name1] <- list(df)
    output_list[name2] <- list(cents)
    start <- start + 1
  }

  ### Find best model
  output_vector <- c()
  start <- start - 1
  for (i in 1:start){
    name <- paste0('df', i)
    final_df <- as.data.frame(output_list[name])
    colnames(final_df) <- append(c('x1', 'x2', 'cl'), colnames(target_df))
```

```
    output_vector[i] <- final_df %>%
      group_by(cl) %>% summarise(n = n()) %>%
      summarise(cal = max(n) - min(n)) %>% pull()
  }

  ### Make plot by best model
  best_num <- which.min(output_vector)
  new_df <- as.data.frame(output_list[paste0('df', best_num)])
  colnames(new_df) <- append(c('x1', 'x2', 'cl'), colnames(target_df))
  new_cents <- as.data.frame(output_list[paste0('cents', best_num)])
  colnames(new_cents) <- c('cl', 'x1', 'x2')

  ori <- output_list$df %>%
    ggplot(aes(x = x1, y = x2, col = cl)) +
    geom_point(shape = 1) + theme_bw() +
    geom_point(data = output_list$cents, shape = 4, col = 'red') +
    ggtitle('Plot of original data') + theme(plot.title = element_text(hjust=0.5))

  new <- new_df %>%
    ggplot(aes(x = x1, y = x2, col = cl)) +
    geom_point(shape = 1) + theme_bw() +
    geom_point(data = new_cents, shape = 4, col = 'red') +
    ggtitle('Plot of best model data') + theme(plot.title = element_text(hjust=0.5))

  print(plot_grid(ori, new, nrow = 2))
  return (new_df)
}
```

```
synth.data <- data.frame(x1 = c(rnorm(20, 3, 1.5), rnorm(20, 0, 1), rnorm(20, 5, 1)),
                         x2 = c(rnorm(20, 0, 1), rnorm(20, 4, 1), rnorm(20, 5, 1)))

synth.data <- k_means_algo(synth.data, x1, x2, 3)
```



k-means 알고리즘을 직접 코드로 구현하는 과정은 다음과 같다.

1. 전체 프로세스를 반복해줄 수 있는 무한반복문 만들기
2. 각 군집과 데이터들 사이에 거리를 구해주기
3. 구해준 거리에 대해 가장 가까운 군집으로 매칭시키기
4. 새롭게 구해진 군집에 대해 평균을 계산하여 중심점 조정하기
5. 1~4번 과정을 반복하면서, break 할 수 있는 조건문 만들기

3-1. 좋은 군집이란? - Total Within Sum of Squares (WSS)

지도학습과 다르게, 비지도학습은 사람들의 주관이 들어가는 경우가 많다.

k-means 알고리즘에서도 k의 값에 따라 결과가 많이 달라진다는 것을 쉽게 알 수 있다.

따라서 군집을 잘 만들었을까를 평가하는 지표가 몇 가지 필요하다.

가장 먼저는 **Total Within Sum of Squares (WSS)**가 있다.

Within sum of squares는 각 중심점으로부터 거리 제곱의 평균을 계산하는 것이다.

모든 군집에 대해 Within sum of squares를 더한 것이 Total Within Sum of Squares 즉, WSS이다.

이 값이 낮을수록 중심점(Centroid)에 더 가까이 모여있기 때문에 좋은 군집이라고 할 수 있다.

하지만 단점으로는 같은 알고리즘을 사용할 때, 군집의 개수가 클수록 WSS는 작아지는 한계가 있다.

```
# WSS
sqrt.edist <- function(u, v){sum((u-v)**2)}

WSS <-
  sapply(1:k, function(i){
    sum(
      apply(split(synth.data, synth.data$cl) [[i]] [, 1:ndim], 1, function(x){
        sqrt.edist(x, cents[i, -1])
      }))
  })

WSS
```

```
## [1] 584.24661  22.28681 461.46292
```

```
sum(WSS)
```

```
## [1] 1067.996
```

```
head(split(synth.data, synth.data$cl) [[1]] [, 1:ndim], 1) # 1번 군집에 해당된 data point
```

```
##      x1        x2
## 1 2.488 -1.424858
```

```
head(cents[1, -1]) # 1번 군집의 Centroid 정보
```

```
##          x1      x2
## 1 0.1346088 4.23744
```

split(synth.data, synth.data$cl) [[i]] [, 1:ndim] 코드에서 i는 군집에 따른 데이터프레임이다.
즉, 현재 list에는 각 군집 별로의 데이터프레임이 담겨있는데, 거기서 x1과 x2만 추려온다는 것이다.
그 값에 대하여 centroid를 담고 있는 cents[i, -1]에 대하여 거리를 계산해준다.
그렇게 되면, WSS에는 k개의 군집 개수 만큼 거리 제곱의 평균 값이 계산된다.

3-2. 좋은 군집이란? - Calinski Harabasz index (CH index)

앞서 이야기했던 것처럼 WSS는 군집의 개수가 클수록 그 값이 필연적으로 작아진다는 한계가 있다.
다시 말하면, 데이터의 개수가 50개 일 때, 군집의 개수가 50개라면 WSS의 값은 0이 된다는 것이다.
따라서 이러한 WSS의 한계점을 보완하고자 CH index라는 개념이 등장했다.
n이 각 데이터의 개수이고, k가 군집의 개수라고 할 때 **CH index = (BSS(k)/(k-1)) / (WSS(k)/(n-k))**

- WSS: 군집 내 분산 - 군집의 중심점과 각 데이터들 사이에 거리 제곱의 합
- TSS: 전체 데이터의 분산 - 모든 데이터의 중심점과 각 데이터들 사이에 거리 제곱의 합
- BSS: 군집 간 분산 - 전체 데이터의 분산(TSS)에서 군집 내 분산(WSS)을 빼준 것

```
# CH-index
wss <- sum(WSS)

all.center <- colMeans(synth.data[, 1:ndim]) # 모든 데이터에 대한 중심점
tss <- sum(
  apply(synth.data[, 1:ndim], 1, function(x){sqrt.edist(x, all.center)}))

bss <- tss - wss
ch.index <- (bss/(k-1)) / (wss/(ndata-k))
ch.index
```

```
## [1] -10.78256
```

n개의 모든 데이터에 대한 중심점을 정해준 후, 그 점과 각 데이터들 사이에 거리의 합을 구한다.
그 값을 tss라고 할때, tss에서 wss를 빼주면, bss가 계산된다.
bss는 각 클러스터들 사이에 거리라고 할 수 있다.

좋은 군집의 기준

1. WSS가 작을수록 좋다. (군집 안에서 서로 모여 있기 때문이다.)
    - k의 값이 커질수록 감소한다.
2. BSS가 클수록 좋다. (각 군집마다 서로 명확히 구분되기 때문이다.)
    - k의 값이 커질수록 증가한다.
3. CH index가 클수록 좋다.

4-1. Hierarchical Clustering

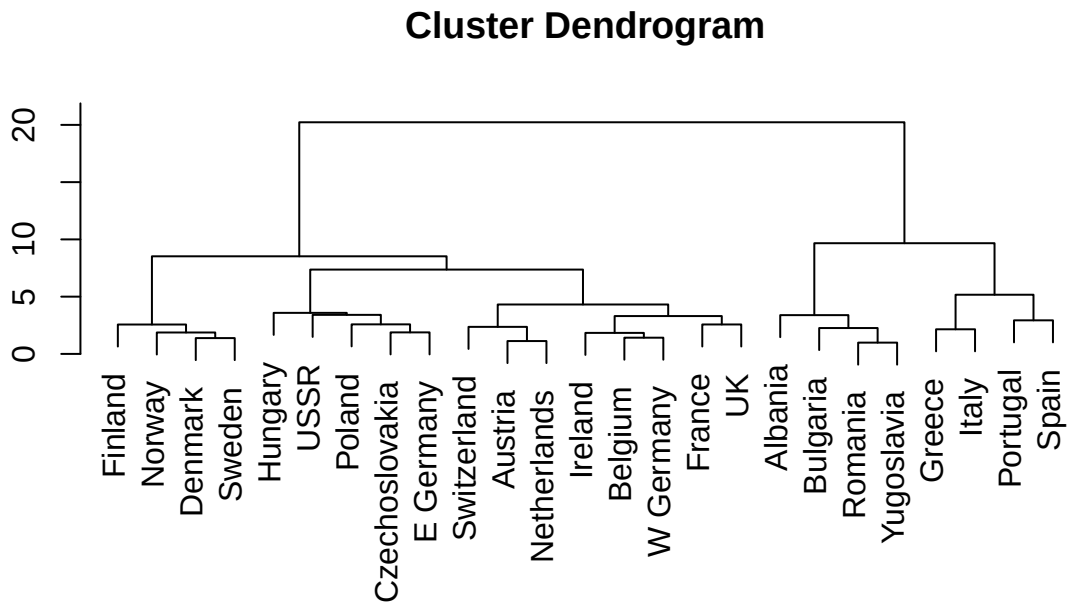
```
path <- 'https://raw.githubusercontent.com/Paul-scpark/Data_Mining_Practicum/main/data/'
protein <- read.table(paste0(path, 'protein.txt'), sep = '\t', header = T)

head(protein)
```

```
##          Country RedMeat WhiteMeat Eggs Milk Fish Cereals Starch Nuts Fr.Veg
## 1        Albania    10.1       1.4  0.5  8.9  0.2    42.3    0.6  5.5    1.7
## 2        Austria     8.9      14.0  4.3 19.9  2.1    28.0    3.6  1.3    4.3
## 3        Belgium    13.5       9.3  4.1 17.5  4.5    26.6    5.7  2.1    4.0
## 4       Bulgaria     7.8       6.0  1.6  8.3  1.2    56.7    1.1  3.7    4.2
## 5 Czechoslovakia     9.7      11.4  2.8 12.5  2.0    34.3    5.0  1.1    4.0
## 6        Denmark    10.6      10.8  3.7 25.0  9.9    21.9    4.8  0.7    2.4
```

```
var.to.use <- colnames(protein)[-1]
pmatrix <- scale(protein[, var.to.use]) # Z 정규화
pcenter <- attr(pmatrix, 'scaled:center')
pscale <- attr(pmatrix, 'scaled:scale')
```

데이터를 불러온 후, 데이터의 전반적인 scale에 차이가 있어서 Z 정규화를 해준다.

```
d <- dist(pmatrix, method = 'euclidean') # 거리 matrix 만들기
pfit <- hclust(d, method = 'ward.D')
plot(pfit, labels = protein$Country, ylab = '', xlab = '', sub = '')
```



그리고 각 데이터들 사이에 거리 matrix를 만들어주고, Hierarchical 군집을 만들어준다.

위의 결과와 같이 Hierarchical Clustering은 nested clusters 형태로 결과를 반환한다.

또한 tree 형태의 diagram (dendrogram)으로 결과가 나오므로, 직관적 해석이 가능하다.

https://iq.opengenus.org/hierarchical-clustering/

4-2. Linkage Methods of Clustering

1. Single Linkage - Minimum distance
   - 각 군집의 데이터에 대해 가장 가까이에 있는 두 개의 data point의 거리를 구한다.
   - 따라서 면적이 넓어질수록 두 점이 가깝게 되니까 좋다고 할 수 있다.
2. Complete Linkage - Maximum distance
   - 각 군집의 데이터에 대해 가장 멀리 있는 두 개의 data point의 거리를 구한다.
   - 따라서 면적이 넓어질수록 불리하다고 할 수 있다.
3. Average Linkage - Average distance
   - 모든 점들 사이에 평균 값을 계산하는 방식이라고 할 수 있다.
   - 따라서 계산량이 가장 많은 방식이다.

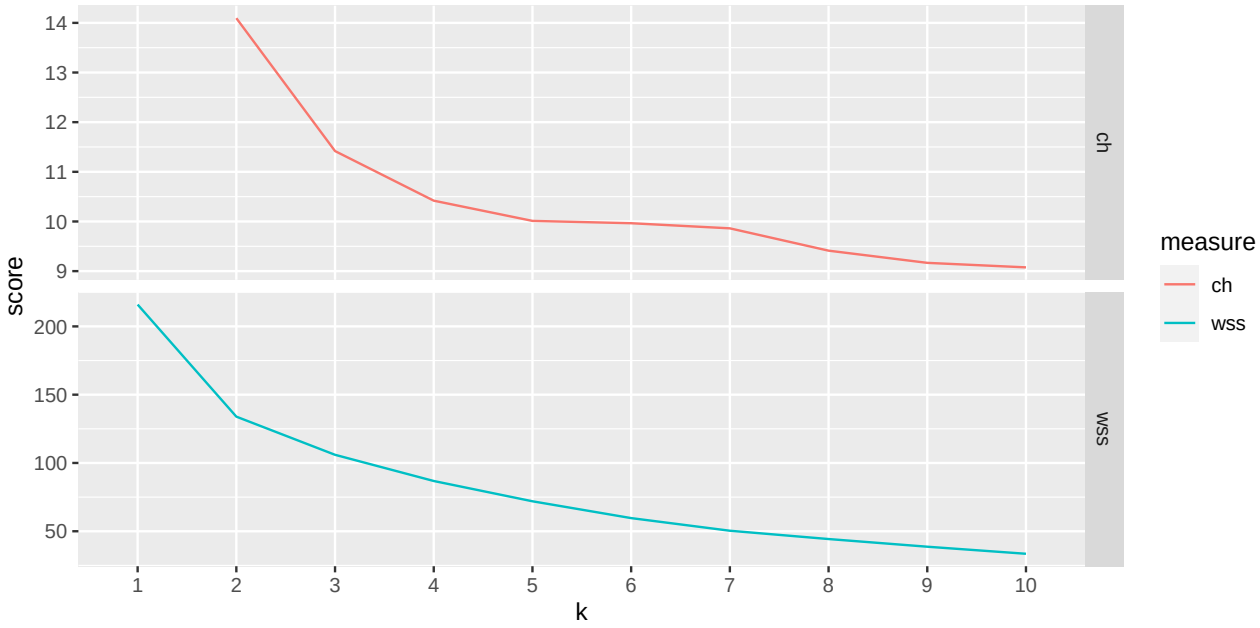
```
groups <- cutree(pfit, k = 5)

print_clusters <- function(labels, k){
  for (i in 1:k){
    print(paste('cluster', i))
    print(head(protein[labels == i, c('Country', 'RedMeat', 'Fish', 'Fr.Veg')], 3))
  }
}

print_clusters(groups, 5)
```

```
## [1] "cluster 1"
##     Country RedMeat Fish Fr.Veg
## 1   Albania    10.1  0.2    1.7
## 4  Bulgaria     7.8  1.2    4.2
## 18  Romania     6.2  1.0    2.8
## [1] "cluster 2"
##   Country RedMeat Fish Fr.Veg
## 2 Austria     8.9  2.1    4.3
## 3 Belgium    13.5  4.5    4.0
## 9  France    18.0  5.7    6.5
## [1] "cluster 3"
##           Country RedMeat Fish Fr.Veg
## 5  Czechoslovakia     9.7  2.0    4.0
## 7       E Germany     8.4  5.4    3.6
## 11        Hungary     5.3  0.3    4.2
## [1] "cluster 4"
##    Country RedMeat Fish Fr.Veg
## 6  Denmark    10.6  9.9    2.4
## 8  Finland     9.5  5.8    1.4
## 15  Norway     9.4  9.7    2.7
## [1] "cluster 5"
##     Country RedMeat Fish Fr.Veg
## 10   Greece    10.2  5.9    6.5
## 13    Italy     9.0  3.4    6.7
## 17 Portugal     6.2 14.2    7.9
```

각 군집에 해당하는 데이터를 출력해보면, 위와 같은 결과를 얻을 수 있다.

5. Picking proper K

```
path <- 'https://raw.githubusercontent.com/Paul-scpark/Data_Mining_Practicum/main/data/'
protein <- read.table(paste0(path, 'protein.txt'), sep = '\t', header = T)
var.to.use <- colnames(protein)[-1]
pmatrix <- scale(protein[, var.to.use]) # Z 정규화

total_df <- data.frame()
for (k in 1:10){
  pclusters <- kmeans(pmatrix, k, nstart = 100, iter.max = 100)

  BSS_output <- pclusters$betweenss
  WSS_output <- pclusters$tot.withinss
  CH_output <- (BSS_output / (k-1)) / (WSS_output / (nrow(pmatrix) - k))

  total_df <- rbind(total_df, c(k, CH_output, WSS_output))
}

colnames(total_df) <- c('k', 'ch', 'wss')
total_df$ch <- ifelse(total_df$ch == -Inf, NA, total_df$ch)

total_df %>%
  gather(measure, score, 2:3) %>%
  ggplot(aes(x = factor(k), y = score, fill = measure, group = 1)) +
  geom_line(aes(color = measure)) +
  facet_grid(measure ~., scales = 'free_y') +
  xlab('k')
```



protein 데이터로 1부터 10까지의 kmeans 알고리즘을 적용하여 ch index와 wss를 확인한다.
WSS는 작고, Ch index는 큰 것이 좋은 군집이므로, k가 5, 6 정도가 가장 좋다고 할 수 있다.

6-1. Density-Based Spatial Clustering of Application with noise - DBSCAN

DBSCAN 알고리즘은 밀도 기반의 알고리즘이라고 할 수 있다.

즉, 어느 점을 기준으로 반경 범위 내에 데이터가 일정 수준 있다면, 하나의 군집으로 인식한다.

1. 특정 범위(Epsilon) 안에 몇 개의 점이 있니? = 밀도를 고려함.
2. 그 범위 안에 최소 데이터 개수(MinPts)를 만족하면, Core point로 인식함.
3. 이 점들은 군집 안에 있는 점들로 border point와 noise point로 구분됨.
    - Border point: Core point의 주위에 있으면서 Epsilon 내에 MinPts 보다 적은 점들
    - Noise point: Core point와 border point가 아닌 점들

6-2. DBSCAN - Algorithm

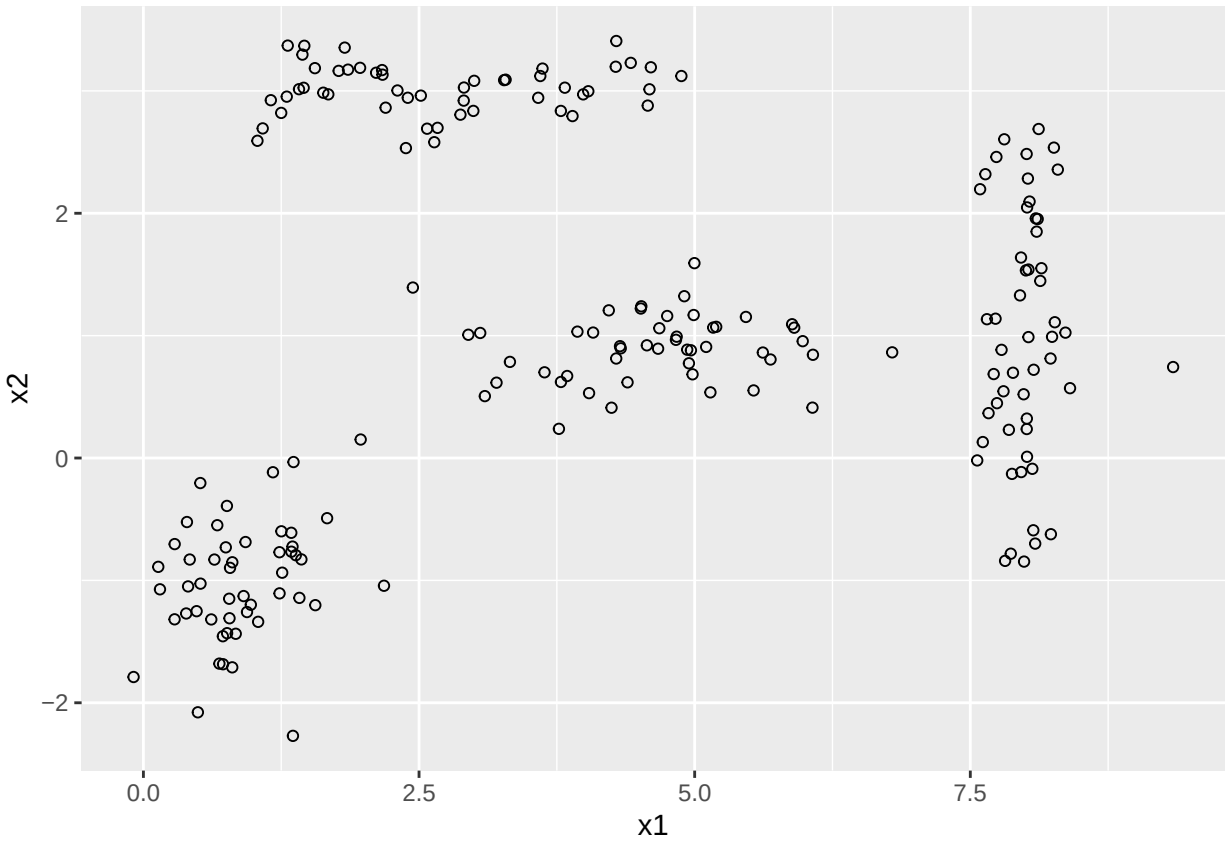
1. Cluster_count를 0으로 설정하고, 모든 점 p에 대하여 다음 과정을 수행한다.
2. 만약 p가 core point가 아니면, null label을 할당한다.
3. 만약 p가 core point라면, 새로운 Cluster를 생성한다.
    - 이 때, Cluster_count를 1 증가시킨다.
    - 밀도 기반으로 p 근방에 있는 데이터에 대하여 군집 여부를 확인한다.
4. 모든 데이터들에 대하여 위 과정을 계속 반복한다.

6-3. DBSCAN 알고리즘의 장단점

1. (+) k-means 알고리즘과 다르게 알고리즘이 자동으로 클러스터의 개수를 찾는다.
2. (+) 선형적인 모양 외에 비선형적인 클러스터의 모양을 갖는다.
3. (+) 노이즈 데이터도 분석할 수 있기 때문에 Outlier 제거에 용이하다.
4. (-) 알고리즘이 이용하는 거리 측정 방법에 따라 결과가 변화한다.
5. (-) 데이터 특성을 잘 모를 경우, 적절한 hyper-parameter 설정이 어렵다.

6-4. DBSCAN - code

```
##### DBSCAN 알고리즘 구현 (전체)
synth.data2 <- data.frame(x1 = c(runif(50, 1, 5), rnorm(50, 1, 0.5),
                                 rnorm(50, 5, 1.5), rnorm(50, 8, 0.2)),
                          x2 = c(rnorm(50, 3, 0.2), rnorm(50, -1, 0.5),
                                 rnorm(50, 1, 0.3), runif(50, -1, 3)))

synth.data2 %>%
  ggplot(aes(x = x1, y = x2)) +
  geom_point(shape = 1)
```



DBSCAN의 결과를 확인하기 위해서 임의의 데이터를 만들고, 그래프를 그려보도록 한다.

```
Eps <- 0.5
MinPts <- 6
ClusterCount <- 1
synth.data2$num <- rep(1:nrow(synth.data2))
synth.data2$cl <- NA

db <- dbscan(synth.data2[, 1:2], Eps, MinPts)
dbscan_plot <- cbind(synth.data2, db$cluster) %>%
  ggplot(aes(x = x1, y = x2, col = factor(db$cluster))) +
  geom_point(shape = 1) +
  ggtitle('Output of DBSCAN package') +
  theme(plot.title = element_text(hjust=0.5))
```

그리고 Epsilon 값을 0.5로, MinPts 값을 6으로 하는 DBSCAN 결과를 확인해보려고 한다.

직접 구현한 알고리즘과 패키지의 결과를 확인하기 위해서 dbscan_plot에 미리 할당해둔다.

```
for (p in 1:nrow(synth.data2)){
  target <- synth.data2[p, ] # 임의로 시작점을 선택

  # 클러스터가 배정이 안된 경우에 실시
  if (is.na(target$cl)){
    # 시작점(target)으로부터 거리를 구해서, Epsilon 보다 작은 데이터만 추리기
    target_df <- synth.data2 %>%
      mutate(dist = sqrt(((target$x1 - x1)**2) + ((target$x2 - x2)**2))) %>%
```

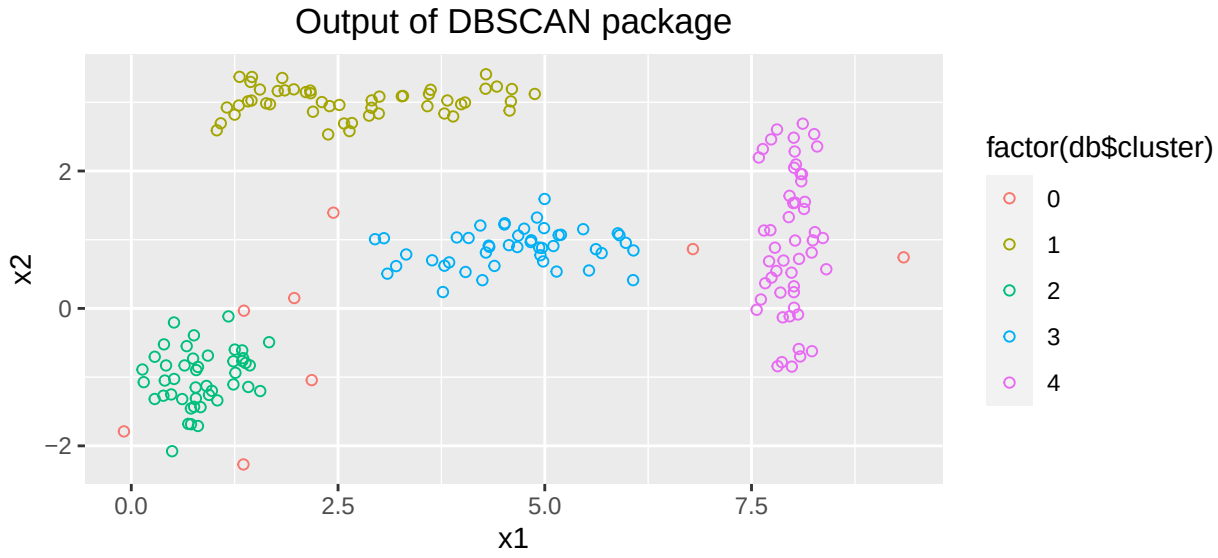
```
      filter(dist <= Eps)
    # 추려진 데이터가 MinPts 개수보다 많은 경우
    if (nrow(target_df) >= MinPts){

      # 새로운 점이 추가되지 않을 때까지, 계속 반복
      # 처음으로 추린 데이터에 대하여 Eplison 범위 안에 점들을 계속해서 추가
      while (TRUE){
        ori <- nrow(target_df)
        # 추려진 데이터(target_df)에 대해 계속해서 Eplison 범위 안에 점들 찾기
        for (i in 1:nrow(target_df)){
          target <- target_df[i, ]
          new_df <- synth.data2 %>%
            mutate(dist = sqrt(((target$x1 - x1)**2) + ((target$x2 - x2)**2))) %>%
            filter(dist <= Eps)
          target_df <- rbind(target_df, new_df)}
        target_df <- target_df[!duplicated(target_df$num), ]
        new <- nrow(target_df)
        if (ori == new){break}} # for문 실행 전과 후의 데이터 개수가 동일하면, break

      # 추려진 데이터에 대해 이웃의 점 개수(neighbor) 구하기
      row.names(target_df) <- NULL
      for (i in 1:nrow(target_df)){
        target <- target_df[i, ]
        neighbor_df <- target_df %>%
          mutate(dist = sqrt(((target$x1 - x1)**2) + ((target$x2 - x2)**2))) %>%
          filter(dist <= Eps)
        target_df[i, 'neighbor'] <- nrow(neighbor_df)}

      # 이웃의 점 개수가 MinPts 보다 큰 점들을 core point로 정의
      # core point 점들에 대해서 Epsilon 범위 내에 모든 점을 같은 Cluster로 labeling
      core_df <- target_df[target_df$neighbor >= MinPts, ]
      row.names(core_df) <- NULL
      for (i in 1:nrow(core_df)){
        target <- core_df[i, ]
        # Core point에 대해 Epsilon 범위 내에 점들을 같은 Cluster로 배정
        final_df <- target_df %>%
          mutate(dist = sqrt(((target$x1 - x1)**2) + ((target$x2 - x2)**2))) %>%
          filter(dist <= Eps)
        synth.data2[final_df$num, 'cl'] <- ClusterCount}
      ClusterCount <- ClusterCount + 1} # Cluster 배정이 마무리되면, Cluster를 1 더해주기
  }
}
```

DBSCAN 알고리즘이 작동하는 순서는 다음과 같다.

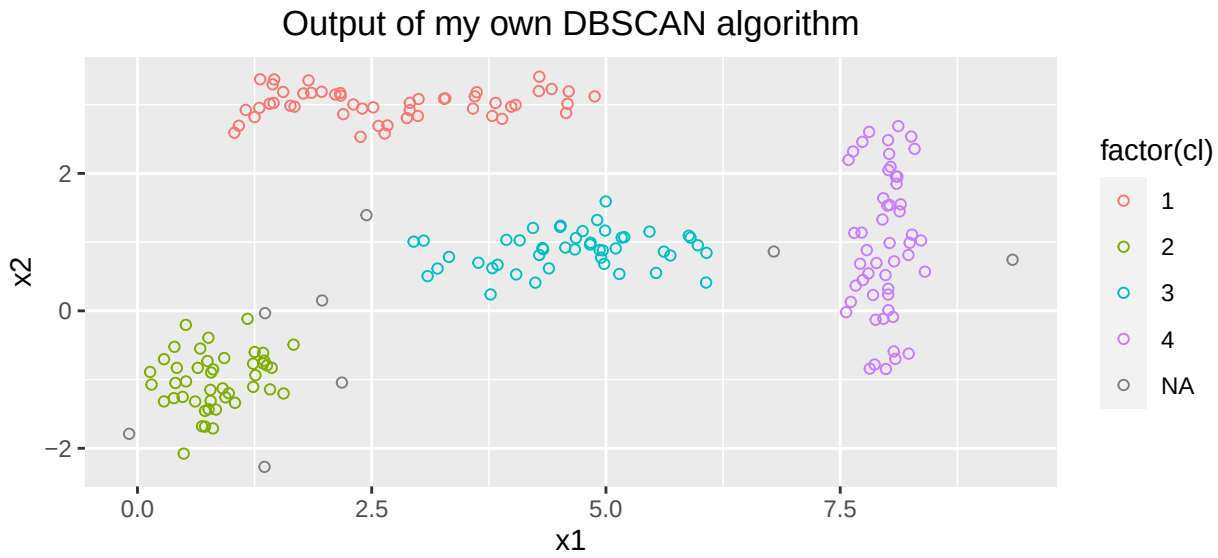
1. 임의의 한 점을 선택하여, 주어진 범위(Epsilon)에 해당하는 이웃 점 개수를 찾는다.
2. 이웃의 점 개수가 MinPts 보다 크면, Core point로 정의한다.
3. Core point는 아니지만, Core point 점의 범위 내에 있으면 Border point로 정의한다.
4. Core point와 Border point는 같은 클러스터로 할당하고, 이외 것은 Noise point로 정의한다.
5. 모든 점을 확인하면서, 클러스터를 생성해주도록 한다.

이와 같은 작동 방식을 살려서 DBSCAN 알고리즘을 직접 구현해보았다.

```
algo_plot <- synth.data2 %>%
  ggplot(aes(x = x1, y = x2, col = factor(cl))) +
  geom_point(shape = 1) +
  ggtitle('Output of my own DBSCAN algorithm') +
  theme(plot.title = element_text(hjust=0.5))

dbscan_plot
```




```
algo_plot
```



직접 구현한 모델을 통해 나온 결과를 algo_plot에 저장해둔다.

그리고 앞서 정의한 dbscan 패키지의 결과와 함께 그래프를 그려본다.

그 결과를 비교해보면, 두 그래프가 비슷하게 나오는 것을 확인할 수 있다.