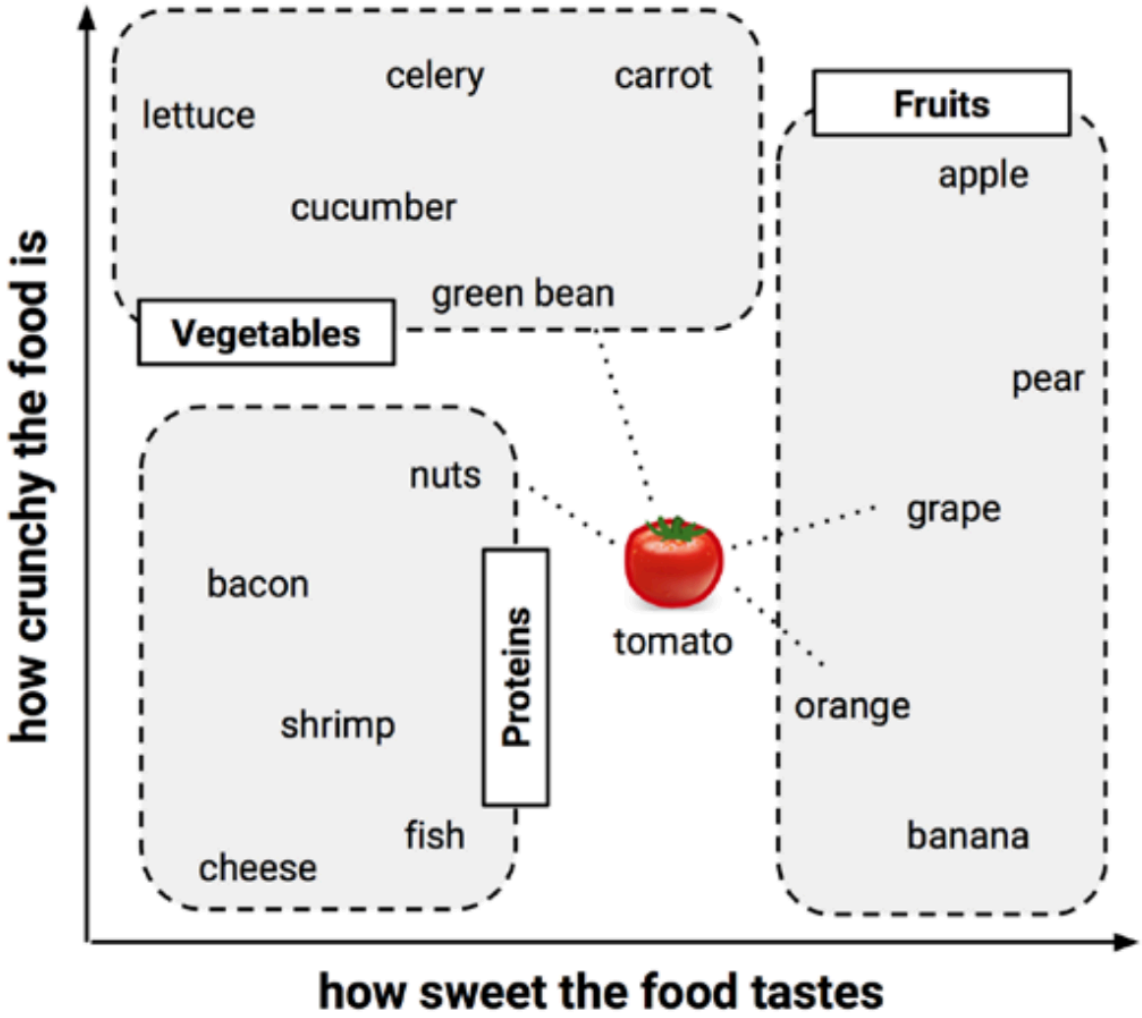
k Nearest Neighbor

Jaeyoon Han

k Nearest Neighbor

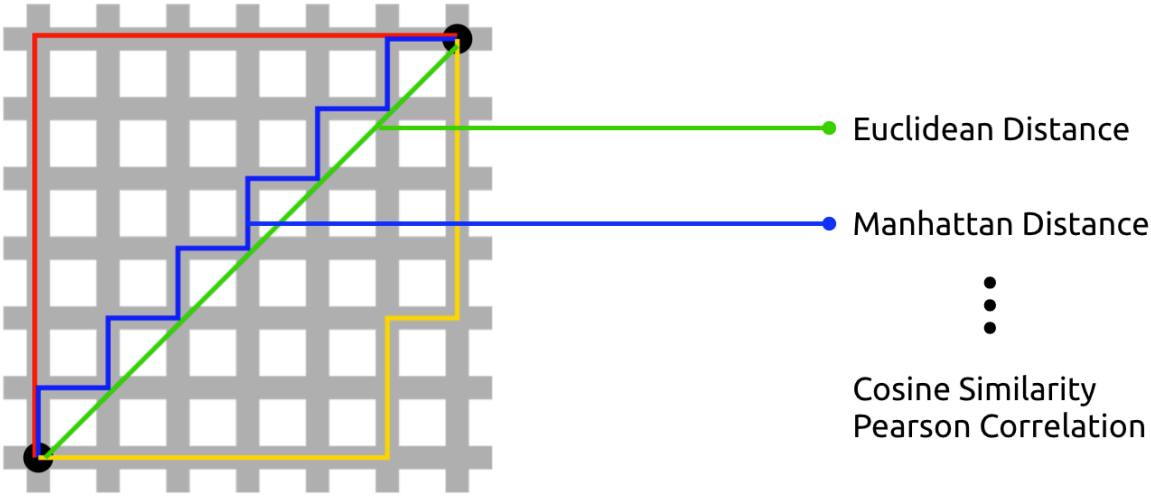
k-최근접 이웃(k Nearest Neighbor)은 유명한 분류 모델 중 하나입니다. 트레이닝 데이터셋이 다양한 범주를 가지고 있다고 가정해봅시다. 어디에 속하는 지 모르는 새로운 데이터에 대해서 가장 유사한 범주로 분류시키는 모델이 바로 k-최근접 이웃입니다.

k-최근접 이웃은 매우 단순합니다. 여러 음식들을 단맛과 아삭함을 기준으로 2차원 평면에 나타내보도록 하겠습니다.



이 때, 기존의 음식들을 채소, 과일, 단백질류 세 범주로 나눌 수 있습니다. 범주 내의 음식들은 서로 유사성이 있기 때문에 서로 간의 거리가 매우 가깝습니다. 여기에 토마토를 중간에 넣어보겠습니다. 넣는 위치는 단맛의 정도와 아삭함의 정도에 기반을 둡니다. 토마토를 세 범주 중 하나로 분류하는 방법은 간단합니다. 가장 가까운 k개의 데이터를 확인하기만 하면 됩니다. 그림에서는 포도와 오렌지, 호두와 콩이 가장 가까운 네 개의 데이터입니다. 따라서 이 모델에서는 토마토를 과일로 분류합니다.

k-최근접 이웃에서 가장 중요한 요소는 거리와 k값입니다. 새로운 데이터가 기존의 데이터와 얼마나 유사한 지 확인하기 위해서는 거리값을 사용합니다. 이 때, 일반적으로 **유클리드 거리(Euclidean Distance)**를 사용합니다. 우리가 평소에 사용하는 거리 개념입니다. 가끔은 **맨하탄 거리(Manhattan Distance)**를 사용하기도 합니다. 실제 맨하탄 거리가 사각형으로 잘 정돈되어 있다는 특징에서 나온 개념입니다. 이 개념들 외에도 **코사인 유사도(Cosine Similarity)**나 **피어슨 상관계수(Pearson Correlation)** 등을 사용합니다.



k값은 새로운 데이터를 기존 데이터와 몇 개를 비교할 지 정하는 값이다. 이 때, k값은 범주 개수의 배수로 두지 않는 것이 좋다. 동점이 나올 수도 있기 때문이죠. 예를 들어서 $k = 10$인 경우에, 두 범주에서 각각 다섯 개의 데이터와 가깝다고 한다면 새로운 데이터의 범주를 선택하기에 문제가 생기죠. 따라서 이런 경우를 배제하기 위하여 k값을 범주 개수의 배수로 두지 않습니다.

또한 너무 작은 k값과 너무 큰 k값을 설정하는 것도 문제가 됩니다. 만약 k값이 너무 작으면 노이즈에 많은 영향을 받고, 과적합이 발생할 수도 있죠. 하지만, k값이 너무 크면 처음부터 데이터가 많이 속해있는 범주로 수렴할 수 밖에 없습니다. 경계가 불분명해지기도 하구요. 따라서 우리는 적당한 k값을 찾아서 모델을 구축해야 합니다.

하지만, 이 k값을 설정하는 방법은 정해진 바가 없습니다. 어림잡기 방법(Rule of Thumbs)을 사용하는 경우도 있지만, 결국은 선험적인 방법을 통해서 설정하게 됩니다.

k-최근접 이웃 알고리즘은 **게으른 학습(Lazy Learning)**의 일종입니다. 데이터를 학습한다기 보다는 저장해놓고, 다른 데이터가 들어오길 기다리기 때문입니다. 따라서 훈련 단계는 굉장히 빠르지만, 실제 예측 과정은 상대적으로 느린 축에 속합니다. 다른 알고리즘들은 대부분 **열정적인 학습(Eager Learning)**에 속하죠.

k-최근접 이웃 알고리즘은 단순하고 효율적이며 데이터 분산에 대한 추가적인 추정이 필요가 없다는 장점이 있습니다. 또한 위에서 말했듯이 훈련 시간도 빠르죠. 하지만 실제 예측과정이 많이 느립니다. 데이터 포인트 사이의 거리를 구해야 되기 때문이죠. 또한 명목형 데이터와 결측 데이터에 대해서는 추가적인 처리가 필요합니다.

실습 : 유방암 진단 (Breast Cancer Wisconsin Diagnostic)

사용할 데이터셋은 유방암 진단 데이터입니다. 디지털화된 유방 덩어리의 미세 침흡인물의 이미지로부터 측정된 일련의 데이터가 포함되어 있죠.

```
wbcd <- read.csv("wisc_bc_data.csv", stringsAsFactors = FALSE)
str(wbcd)
```

```
'data.frame':   569 obs. of  32 variables:
 $ id               : int  87139402 8910251 905520 868871 9012568 906539 925291 87880 862989 89827 ...
 $ diagnosis        : chr  "B" "B" "B" "B" ...
 $ radius_mean      : num  12.3 10.6 11 11.3 15.2 ...
 $ texture_mean     : num  12.4 18.9 16.8 13.4 13.2 ...
 $ perimeter_mean   : num  78.8 69.3 70.9 73 97.7 ...
 $ area_mean        : num  464 346 373 385 712 ...
 $ smoothness_mean  : num  0.1028 0.0969 0.1077 0.1164 0.0796 ...
 $ compactness_mean : num  0.0698 0.1147 0.078 0.1136 0.0693 ...
 $ concavity_mean   : num  0.0399 0.0639 0.0305 0.0464 0.0339 ...
 $ points_mean      : num  0.037 0.0264 0.0248 0.048 0.0266 ...
 $ symmetry_mean    : num  0.196 0.192 0.171 0.177 0.172 ...
 $ dimension_mean   : num  0.0595 0.0649 0.0634 0.0607 0.0554 ...
 $ radius_se        : num  0.236 0.451 0.197 0.338 0.178 ...
 $ texture_se       : num  0.666 1.197 1.387 1.343 0.412 ...
 $ perimeter_se     : num  1.67 3.43 1.34 1.85 1.34 ...
 $ area_se          : num  17.4 27.1 13.5 26.3 17.7 ...
 $ smoothness_se    : num  0.00805 0.00747 0.00516 0.01127 0.00501 ...
 $ compactness_se   : num  0.0118 0.03581 0.00936 0.03498 0.01485 ...
 $ concavity_se     : num  0.0168 0.0335 0.0106 0.0219 0.0155 ...
 $ points_se        : num  0.01241 0.01365 0.00748 0.01965 0.00915 ...
 $ symmetry_se      : num  0.0192 0.035 0.0172 0.0158 0.0165 ...
 $ dimension_se     : num  0.00225 0.00332 0.0022 0.00344 0.00177 ...
 $ radius_worst     : num  13.5 11.9 12.4 11.9 16.2 ...
 $ texture_worst    : num  15.6 22.9 26.4 15.8 15.7 ...
 $ perimeter_worst  : num  87 78.3 79.9 76.5 104.5 ...
 $ area_worst       : num  549 425 471 434 819 ...
 $ smoothness_worst : num  0.139 0.121 0.137 0.137 0.113 ...
 $ compactness_worst: num  0.127 0.252 0.148 0.182 0.174 ...
 $ concavity_worst  : num  0.1242 0.1916 0.1067 0.0867 0.1362 ...
 $ points_worst     : num  0.0939 0.0793 0.0743 0.0861 0.0818 ...
 $ symmetry_worst   : num  0.283 0.294 0.3 0.21 0.249 ...
 $ dimension_worst  : num  0.0677 0.0759 0.0788 0.0678 0.0677 ...
```

`str()` 함수를 통해서 알아본 결과, 569개의 observation, 32개의 feature로 구성되어 있음을 알 수 있습니다. 각각은 다음과 같습니다.

변수명	설명
id	환자 식별자
radius	반지름

texture	텍스쳐
perimeter	둘레
area	면적
smoothness	평활도
compactness	다짐도
concavity	요면
points	요면점
symmetry	대칭
dimension	프랙탈 차원

이 때, 환자 식별자 `id` 는 랜덤으로 정렬되어 있기 때문에 트레이닝 데이터와 테스트 데이터를 나눌 때 임의대로 나누어도 큰 문제가 되지 않음을 알 수 있습니다. 또한 각 변수의 뒤의 접미사들인 `mean` , `se` , `worst` 는 각각 평균, 표준편차, 가장 나쁜값을 의미합니다.

Data Preprocessing

```
# Remove the attribute 'id'
wbcd <- wbcd[-1]

# The variable that we need to predict
table(wbcd$diagnosis)
```

B	M
357	212

여기서 B는 양성(Benign), M은 악성(Malignant)를 나타냅니다. 즉, M이라고 나타난 부분이 실제 암세포가 있는 환자들입니다. 기존의 데이터 레이블에 정확한 정보를 부여하겠습니다.

```
wbcd$diagnosis <- factor(wbcd$diagnosis, levels = c("B", "M"), labels = c("Benign",
    "Malignant"))

prop.table(table(wbcd$diagnosis))
```

Benign	Malignant
0.6274165	0.3725835

Build a model

```
wbcd_train_labels <- wbcd[1:400, 1]
wbcd_test_labels <- wbcd[401:nrow(wbcd), 1]

wbcd_train <- wbcd[1:400, -1]
wbcd_test <- wbcd[401:nrow(wbcd), -1]
```

처음 kNN을 실행할 때 k의 값을 정하는 것이 가장 큰 문제입니다. 일반적으로 Rule of Thumbs에 의해 전체 데이터의 제곱근으로 잡는 경우가 많습니다. 이 때, 짝수를 설정하면 문제가 생길 수 있는데, 예를 들어 `k = 20` 으로 설정하여 만약 10개씩으로 나눠질 경우 랜덤으로 배정이 됩니다. 모델을 구축할 때마다 결과가 달라지죠. 따라서 일반적으로 홀수로 설정하여 모델을 구축합니다.

```
library(class)  # for kNN
library(caret)

wbcd_pred <- knn(train = wbcd_train, test = wbcd_test, cl = wbcd_train_labels,
    k = 21)

confusionMatrix(wbcd_pred, wbcd_test_labels)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  Benign Malignant
  Benign       106         9
  Malignant      0        54

               Accuracy : 0.9467
                 95% CI : (0.9013, 0.9754)
    No Information Rate : 0.6272
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8827
 Mcnemar's Test P-Value : 0.007661

            Sensitivity : 1.0000
            Specificity : 0.8571
         Pos Pred Value : 0.9217
         Neg Pred Value : 1.0000
             Prevalence : 0.6272
         Detection Rate : 0.6272
   Detection Prevalence : 0.6805
      Balanced Accuracy : 0.9286

       'Positive' Class : Benign
```

Recall = Specificity는 54/63으로 85.7%입니다. 실제 악성인데, 양성으로 나온 경우가 63번 중 9번이나 된다는 것은 다소 치명적이죠. 전체 Accuracy는 94.7%로 높지만, 실제 상황에서는 Specificity가 낮으므로 굉장히 위험한 모델이 됩니다.

Model Improvement

모든 변수에 대해서 값들의 분포가 서로 너무 다릅니다. 이 값들을 모두 normalize해서 0과 1 사이의 숫자로 나타내면 값들의 분포가 균일해질겁니다.

```
normalize <- function(x) {
    return((x - min(x))/(max(x) - min(x)))
}

wbcd_n <- as.data.frame(lapply(wbcd[2:31], normalize))
summary(wbcd_n)
```

radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	points_mean	symmetry_mea
Min. :0.0000	Min. :0.0000	Min. :0.0000	Min. :0.0000	Min. :0.0000	Min. :0.0000	Min. :0.00000	Min. :0.0000	Min. :0.0000
1st Qu.:0.2233	1st Qu.:0.2185	1st Qu.:0.2168	1st Qu.:0.1174	1st Qu.:0.3046	1st Qu.:0.1397	1st Qu.:0.06926	1st Qu.:0.1009	1st Qu.:0.2823
Median :0.3024	Median :0.3088	Median :0.2933	Median :0.1729	Median :0.3904	Median :0.2247	Median :0.14419	Median :0.1665	Median :0.3697
Mean :0.3382	Mean :0.3240	Mean :0.3329	Mean :0.2169	Mean :0.3948	Mean :0.2606	Mean :0.20806	Mean :0.2431	Mean :0.3796
3rd Qu.:0.4164	3rd Qu.:0.4089	3rd Qu.:0.4168	3rd Qu.:0.2711	3rd Qu.:0.4755	3rd Qu.:0.3405	3rd Qu.:0.30623	3rd Qu.:0.3678	3rd Qu.:0.4530
Max. :1.0000	Max. :1.0000	Max. :1.0000	Max. :1.0000	Max. :1.0000	Max. :1.0000	Max. :1.00000	Max. :1.0000	Max. :1.0000

```
wbcd_train <- wbcd_n[1:400, -1]
wbcd_test <- wbcd_n[401:nrow(wbcd), -1]

wbcd_pred2 <- knn(train = wbcd_train, test = wbcd_test, cl = wbcd_train_labels,
    k = 21)

confusionMatrix(wbcd_pred2, wbcd_test_labels)
```

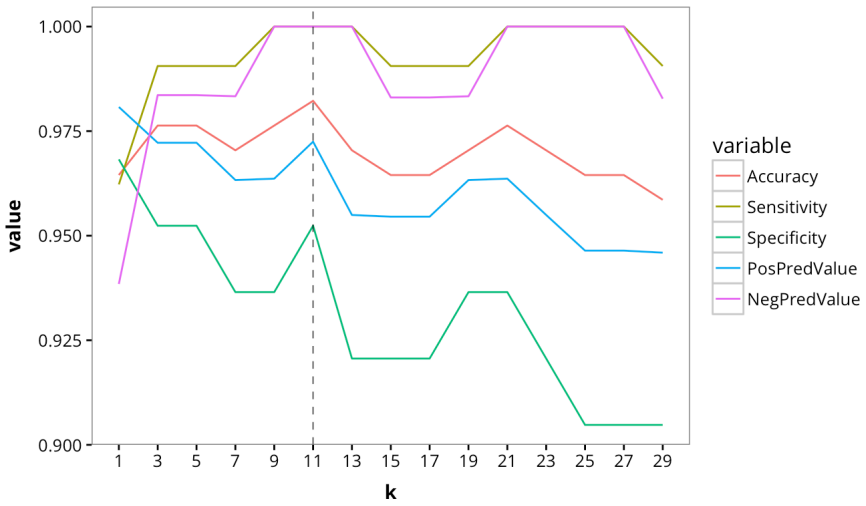
```
Confusion Matrix and Statistics

          Reference
Prediction  Benign Malignant
  Benign       106         4
  Malignant      0        59

               Accuracy : 0.9763
                 95% CI : (0.9405, 0.9935)
    No Information Rate : 0.6272
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9487
 Mcnemar's Test P-Value : 0.1336

            Sensitivity : 1.0000
            Specificity : 0.9365
         Pos Pred Value : 0.9636
         Neg Pred Value : 1.0000
             Prevalence : 0.6272
         Detection Rate : 0.6272
   Detection Prevalence : 0.6509
      Balanced Accuracy : 0.9683

       'Positive' Class : Benign
```

Model Evaluation

```
Result <- data.frame(k = NULL, Accuracy = NULL, Sensitivity = NULL, Specificity = NULL,
    PosPredValue = NULL, NegPredValue = NULL)
sequence <- seq(1, 30, by = 2)
for (i in sequence) {
    pred <- knn(train = wbcd_train, test = wbcd_test, cl = wbcd_train_labels,
        k = i)
    confMat <- confusionMatrix(pred, wbcd_test_labels)
    currentResult <- data.frame(k = i, Accuracy = confMat$overall[1], Sensitivity = confMat$byClass[1],
        Specificity = confMat$byClass[2], PosPredValue = confMat$byClass[3],
        NegPredValue = confMat$byClass[4])
    Result <- bind_rows(Result, currentResult)
}

Result
```

k	Accuracy	Sensitivity	Specificity	PosPredValue	NegPredValue
1	0.9644970	0.9622642	0.9682540	0.9807692	0.9384615
3	0.9763314	0.9905660	0.9523810	0.9722222	0.9836066
5	0.9763314	0.9905660	0.9523810	0.9722222	0.9836066
7	0.9704142	0.9905660	0.9365079	0.9633028	0.9833333
9	0.9763314	1.0000000	0.9365079	0.9636364	1.0000000
11	0.9822485	1.0000000	0.9523810	0.9724771	1.0000000
13	0.9704142	1.0000000	0.9206349	0.9549550	1.0000000
15	0.9644970	0.9905660	0.9206349	0.9545455	0.9830508
17	0.9644970	0.9905660	0.9206349	0.9545455	0.9830508
19	0.9704142	0.9905660	0.9365079	0.9633028	0.9833333
21	0.9763314	1.0000000	0.9365079	0.9636364	1.0000000
23	0.9704142	1.0000000	0.9206349	0.9549550	1.0000000
25	0.9644970	1.0000000	0.9047619	0.9464286	1.0000000
27	0.9644970	1.0000000	0.9047619	0.9464286	1.0000000
29	0.9585799	0.9905660	0.9047619	0.9459459	0.9827586

```
library(reshape2)
moltenResult <- melt(Result, id.vars = "k")

ggplot(data = moltenResult, aes(x = k, y = value, color = variable)) + geom_line() +
    scale_x_continuous(breaks = sequence) + geom_vline(mapping = aes(xintercept = 11),
    linetype = "dashed", alpha = 0.5)
```



Z-scaling + Model Evaluation

```
wbcd_z <- as.data.frame(lapply(wbcd[2:31], scale))
summary(wbcd_z)
```

radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	points_mean	symmetry_mea
Min. :-2.0279	Min. :-2.2273	Min. :-1.9828	Min. :-1.4532	Min. :-3.10935	Min. :-1.6087	Min. :-1.1139	Min. :-1.2607	Min. :-2.74171
1st Qu.:-0.6888	1st Qu.:-0.7253	1st Qu.:-0.6913	1st Qu.:-0.6666	1st Qu.:-0.71034	1st Qu.:-0.7464	1st Qu.:-0.7431	1st Qu.:-0.7373	1st Qu.:-0.7026
Median :-0.2149	Median :-0.1045	Median :-0.2358	Median :-0.2949	Median :-0.03486	Median :-0.2217	Median :-0.3419	Median :-0.3974	Median :-0.07156
Mean : 0.0000	Mean : 0.0000	Mean : 0.0000	Mean : 0.0000	Mean : 0.00000	Mean : 0.0000	Mean : 0.0000	Mean : 0.0000	Mean : 0.00000
3rd Qu.: 0.4690	3rd Qu.: 0.5837	3rd Qu.: 0.4992	3rd Qu.: 0.3632	3rd Qu.: 0.63564	3rd Qu.: 0.4934	3rd Qu.: 0.5256	3rd Qu.: 0.6464	3rd Qu.: 0.5303
Max. : 3.9678	Max. : 4.6478	Max. : 3.9726	Max. : 5.2459	Max. : 4.76672	Max. : 4.5644	Max. : 4.2399	Max. : 3.9245	Max. : 4.48081

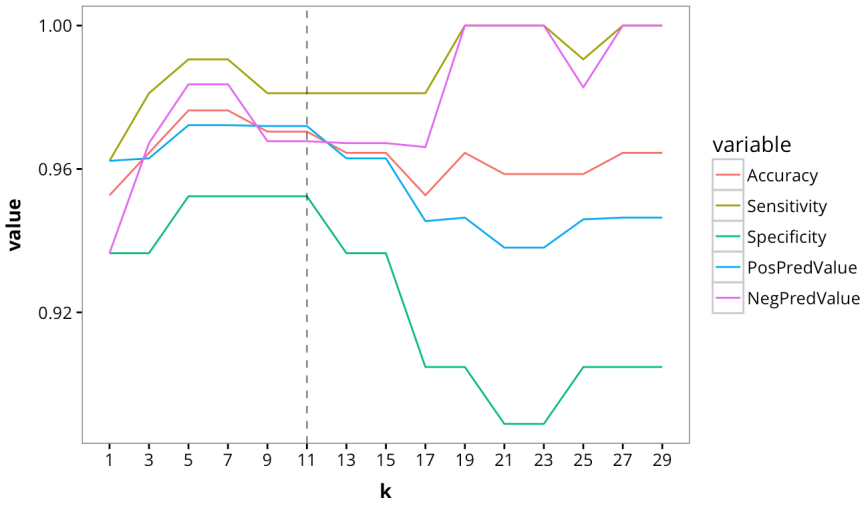
```
wbcd_train_z <- wbcd_z[1:400, -1]
wbcd_test_z <- wbcd_z[401:nrow(wbcd), -1]

wbcd_pred3 <- knn(train = wbcd_train_z, test = wbcd_test_z, cl = wbcd_train_labels,
    k = 21)

confusionMatrix(wbcd_pred3, wbcd_test_labels)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  Benign Malignant
  Benign       106         7
  Malignant      0        56

               Accuracy : 0.9586
                 95% CI : (0.9165, 0.9832)
    No Information Rate : 0.6272
    P-Value [Acc > NIR] : < 2e-16

                  Kappa : 0.9094
 Mcnemar's Test P-Value : 0.02334

            Sensitivity : 1.0000
            Specificity : 0.8889
         Pos Pred Value : 0.9381
         Neg Pred Value : 1.0000
             Prevalence : 0.6272
         Detection Rate : 0.6272
   Detection Prevalence : 0.6686
      Balanced Accuracy : 0.9444

       'Positive' Class : Benign
```

```
Result2 <- data.frame(k = NULL, Accuracy = NULL, Sensitivity = NULL, Specificity = NULL,
    PosPredValue = NULL, NegPredValue = NULL)
sequence <- seq(1, 30, by = 2)
for (i in sequence) {
    pred <- knn(train = wbcd_train_z, test = wbcd_test_z, cl = wbcd_train_labels,
        k = i)
    confMat <- confusionMatrix(pred, wbcd_test_labels)
    currentResult <- data.frame(k = i, Accuracy = confMat$overall[1], Sensitivity = confMat$byClass[1],
        Specificity = confMat$byClass[2], PosPredValue = confMat$byClass[3],
        NegPredValue = confMat$byClass[4])
    Result2 <- bind_rows(Result2, currentResult)
}

Result2
```

k	Accuracy	Sensitivity	Specificity	PosPredValue	NegPredValue
1	0.9526627	0.9622642	0.9365079	0.9622642	0.9365079
3	0.9644970	0.9811321	0.9365079	0.9629630	0.9672131
5	0.9763314	0.9905660	0.9523810	0.9722222	0.9836066
7	0.9763314	0.9905660	0.9523810	0.9722222	0.9836066
9	0.9704142	0.9811321	0.9523810	0.9719626	0.9677419
11	0.9704142	0.9811321	0.9523810	0.9719626	0.9677419
13	0.9644970	0.9811321	0.9365079	0.9629630	0.9672131
15	0.9644970	0.9811321	0.9365079	0.9629630	0.9672131
17	0.9526627	0.9811321	0.9047619	0.9454545	0.9661017
19	0.9644970	1.0000000	0.9047619	0.9464286	1.0000000
21	0.9585799	1.0000000	0.8888889	0.9380531	1.0000000
23	0.9585799	1.0000000	0.8888889	0.9380531	1.0000000
25	0.9585799	0.9905660	0.9047619	0.9459459	0.9827586
27	0.9644970	1.0000000	0.9047619	0.9464286	1.0000000
29	0.9644970	1.0000000	0.9047619	0.9464286	1.0000000

```
moltenResult <- melt(Result2, id.vars = "k")

ggplot(data = moltenResult, aes(x = k, y = value, color = variable)) + geom_line() +
    scale_x_continuous(breaks = sequence) + geom_vline(mapping = aes(xintercept = 11),
    linetype = "dashed", alpha = 0.5)
```



전체적으로 z-scaling보다는 max-min scaling의 결과가 훨씬 좋음을 알 수 있습니다.

Clustering

군집화(Clustering)은 주어진 데이터를 구분할 수 있는 클래스(class)에 대한 지식이 *없는* 상태에서, 각 데이터들의 유사도에 근거하여 데이터를 구분하는 방법론을 말합니다. 지금까지 배웠던 모든 방법론은 **지도 학습(Supervised Learning)**이었던 반면, 군집화는 **비지도 학습(Unsupervised Learning)**입니다.

군집화 기법은 매우 다양합니다. 데이터의 형태나 크기 등에 대하여 서로 다른 알고리즘을 사용하게 됩니다. 오늘은 다양한 알고리즘 중에서 가장 자주 사용하는 세 알고리즘에 대해서 공부하겠습니다.

k-Means

k-Means 알고리즘은 군집화 알고리즘 중에 가장 대중적인 알고리즘입니다. 가장 단순한 알고리즘이죠. 주어진 데이터를 k개의 군집(Cluster)로 묶는 알고리즘입니다. 이 때, 각 클러스터의 중심값이 해당 군집 데이터들의 **평균**으로 구해지기 때문에, 알고리즘 이름이 k-Means Algorithm입니다.

군집의 개수를 설정하고, 평균을 구하는 과정에서는 거리 개념이 들어갑니다. k-최근접 이웃과 유사하죠. 데이터 포인트와 중심점까지의 거리가 최소가 되는 중심점을 찾는 것이 이 알고리즘의 특징입니다. 이 때, 사용되는 방법은 바로 SSE(The Sum of Squared Error)입니다.

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \text{dist}(p, c_i)^2.$$

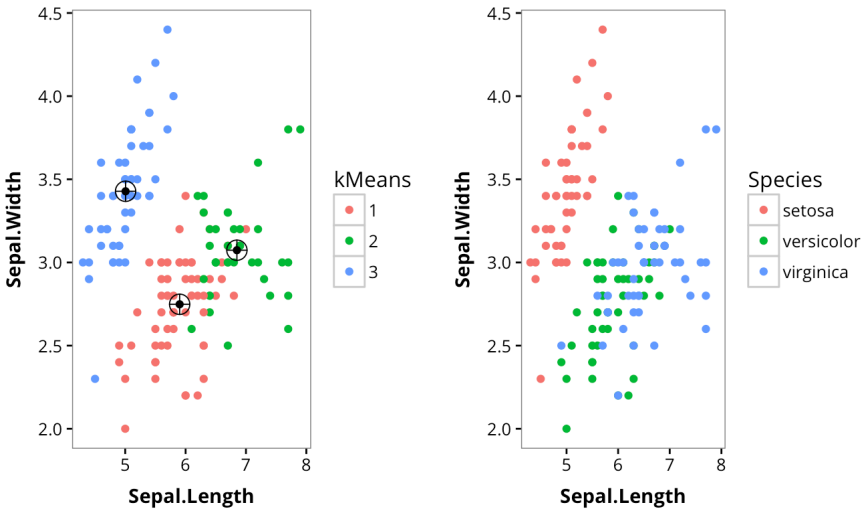
```
library(gridExtra)
data(iris)
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

```
iris_kMeans <- kmeans(iris[1:4], centers = 3)
iris$kMeans <- factor(iris_kMeans$cluster)
kcenters <- data.frame(iris_kMeans$centers)

table(iris$kMeans, iris$Species)
```

/	setosa	versicolor	virginica
1	0	48	14
2	0	2	36
3	50	0	0

```
gg1 <- ggplot() + geom_point(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
    color = kMeans)) + geom_point(data = kcenters, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(data = kcenters, aes(x = Sepal.Length, y = Sepal.Width), size = 5,
        shape = 10)

gg2 <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
    geom_point()

grid.arrange(gg1, gg2, ncol = 2)
```



각 군집화 알고리즘들은 각기 다른 장단점을 가지고 있습니다. k-Means Algorithm의 경우 단순한 알고리즘 덕분에 큰 데이터에도 쉽게 사용할 수 있고, 이에 따라서 확장성이 매우 큽니다(Scalable). 하지만 적절한 k값을 고르는 것이 굉장히 어려우며, 평균값을 사용하기 때문에 노이즈와 이상치에 대해서 매우 약한 모습을 보입니다. 그 뿐만 아니라, 원형 군집(Convex-shaped Data)이 아닌 경우에는 사용하기 어렵습니다.

k-Medoids

k-Medoids는 k-Means와 유사하지만 메인 아이디어가 크게 다릅니다. k-Means의 경우 평균값을 통해서 클러스터의 중심값을 구하기 때문에, 실제 데이터 포인트가 아닌 곳에도 중심이 생길 수 있습니다. 하지만 k-Medoids 알고리즘의 경우, 클러스터의 중심점이 실제 데이터 포인트가 됩니다. 이런 특징 때문에 k-Means 알고리즘과 다르게 노이즈와 이상치에 대해서 매우 견고하다는(Robust) 성질을 가지고 있습니다.

하지만, Medoids를 구할 때 계속해서 다른 데이터 포인트와의 거리를 반복해서 계산해야 되기 때문에 알고리즘의 실행 시간은 데이터가 커지는 정도의 제곱에 비례해서 커지게 됩니다. 따라서 큰 데이터에 대해서는 사용하기 힘들다는 단점이 있죠. 추가적으로 k값을 정하는 문제나 기존의 k-Means가 가지고 있는 단점인 원형 군집에만 사용할 수 있다는 단점도 가지고 있습니다.

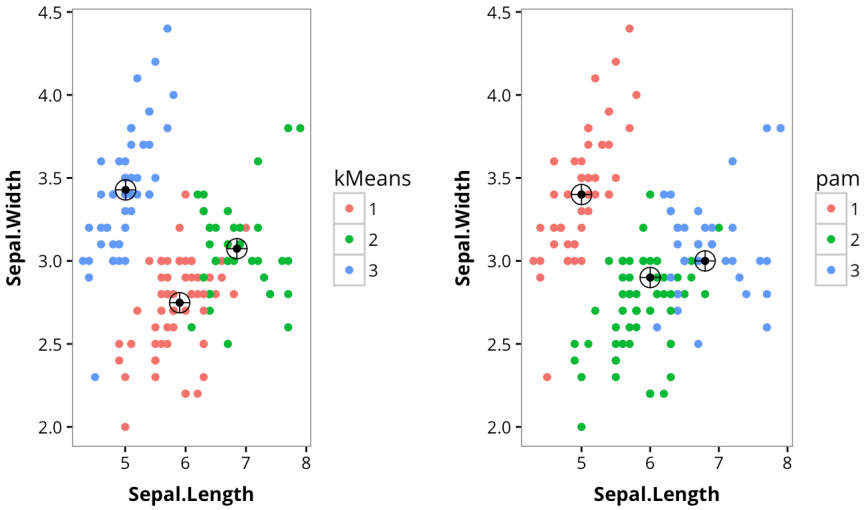
k-Medoids 알고리즘은 k-Means 알고리즘과 다르게 AEC(Absolute-Error Criterion)을 사용합니다.

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \text{dist}(p, o_i)^2.$$

```
library(cluster)
pam_iris <- pam(iris[1:4], k = 3)
iris$pam <- factor(pam_iris$clustering)
pamcenter <- data.frame(pam_iris$medoids)

table(iris$pam, iris$Species)
```

/	setosa	versicolor	virginica
1	50	0	0
2	0	48	14
3	0	2	36

```
gg3 <- ggplot() + geom_point(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
    color = pam)) + geom_point(data = pamcenter, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(data = pamcenter, aes(x = Sepal.Length, y = Sepal.Width), size = 5,
        shape = 10)

grid.arrange(gg1, gg3, ncol = 2)
```

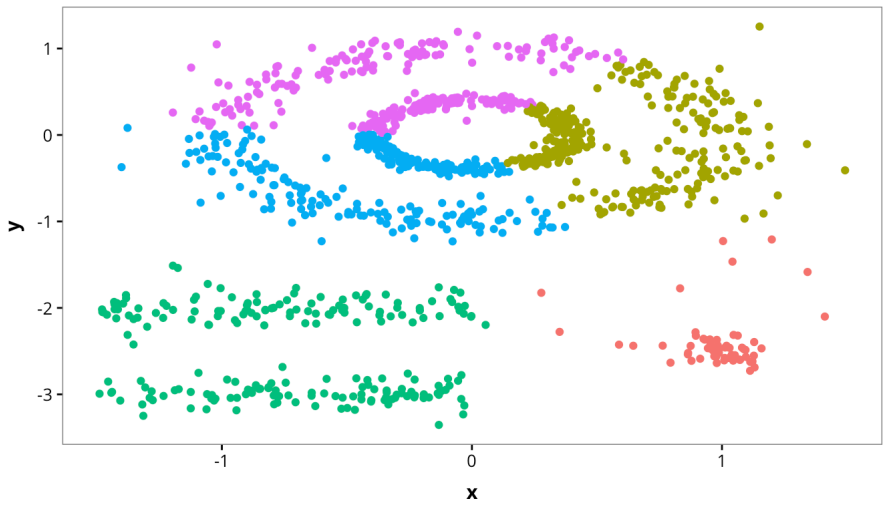


DBSCAN

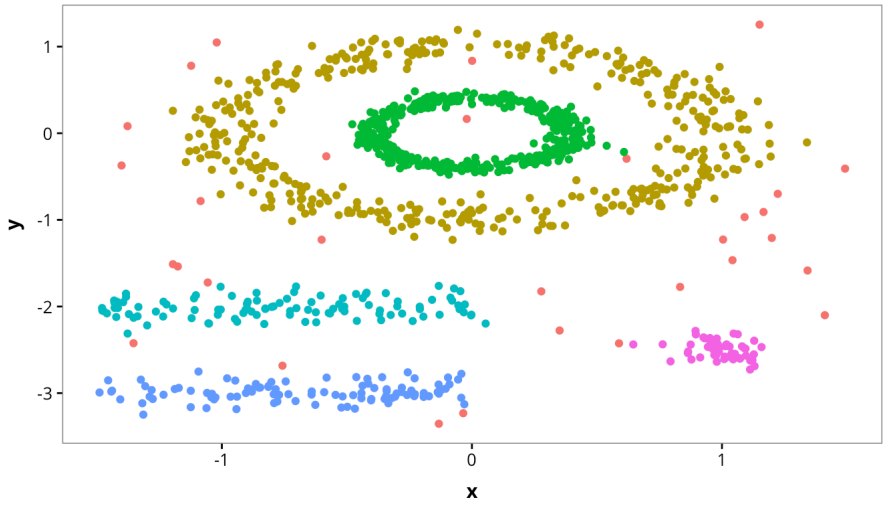
DBSCAN 알고리즘은 위의 두 알고리즘과 많이 다릅니다. 우선 가장 큰 차이점은 원형 군집이 아닌 임의의 형태의 군집이라도 군집화가 가능합니다. 심지어 S자 형태의 군집도 파악해낼 수 있습니다. 이게 가능해진 이유는 DBSCAN 알고리즘의 특징 때문입니다. DBSCAN 알고리즘은 어떤 **핵심 오브젝트(Core object)**를 시작으로 밀도가 높은 이웃 데이터들을 찾아냅니다. 그렇게 묶인 이웃들은 하나의 클러스터가 됩니다. 밀도가 낮은 데이터들은 군집으로 형성되지 못하고 이상치로 남게 됩니다. 정말 밀집되어 있는 클러스터들만 남게 되죠.

하지만 DBSCAN 알고리즘도 단점이 있습니다. 각 데이터들 사이의 계속 반복해서 계산해야되기 때문에 알고리즘 실행 시간이 길어집니다. k-Medoids와 같이 데이터 크기의 제곱에 비례하죠. 그 뿐만 아니라, 알고리즘 실행을 위한 매개변수(Parameters)가 너무 많습니다. 얼마나 가깝고 많은 점들로 구성되어 있어야 '조밀하다(dense)'라고 정의할 수 있을까에 대한 문제죠. 이 때 쓰이는 매개변수는 `eps` 와 `MinPts` 입니다. `eps` 는 최대 거리, `MinPts` 는 최소 점의 개수입니다.

```
library(factoextra)
df <- multishapes[, 1:2]

kmeans_df <- kmeans(df, 5)

km <- ggplot(data = df, aes(x = x, y = y, color = factor(kmeans_df$cluster))) +
    geom_point() + theme(legend.position = "null")
km
```




```
library(fpc)
dbscan <- dbscan(df, eps = 0.15, MinPts = 5)
dbscan_Index <- factor(dbscan$cluster)

db <- ggplot(data = df, aes(x = x, y = y, color = dbscan_Index)) + geom_point() +
    theme(legend.position = "null")
db
```




```
grid.arrange(km, db, ncol = 2)
```