선형 모델의 예측력 or 설명력을 높이기 위해 여러 정규화 방법(여기서 말하는 정규화는 모델을 변형하여 과적합을 완화함으로써 일반화 성능을 높혀주는 기법)을 사용함
⊠ -> 대표적인 shrinkage 방법에는 ridge regression과 lasso가 있음.

1. Ridge regression(능형 회귀)

- Ridge 회귀는 다중공선성을 처리하고, 과적합을 다루며, 계수를 0으로 축소시키지만 Lasso 회귀와 달리 완전히 제거하지 않으면서 모델 복잡도를 줄이는 능력.
- 너무나 많은 학습으로 인해 과적합되는 상황에서 편향을 조금 더 주어 결과의 분산을 낮추는 방법

기본 선형모델을 사용하면 Overfitting이 발생할 수 있는데, 이 경우 데이터에 매우 적합되어 극단적으로 오르락내리락하는 그래프가 생김.
(ex. Y = 2131232 - 42223x + 23042x^2)
-> 이렇게 Variance가 큰 상황을 막기 위해, 계수 자체가 크면 페널티를 주는 수식을 추가한 것이 ridge regression임.
ridge regression은 어떤 값을 통해 이 기울기가 덜 민감하게 반응하게끔 만드는데, 이 값을 람다(lambda, λ)라고 한다.
람다는 값이 커질수록 회귀계수들을 0으로 수렴(기울기를 평균과 비슷하게)시키며 이것은 덜 중요한 특성의 개수를 줄이는 효과이다(과적합을 줄이는 효과), 람다가 0에 가까워질수록 다중회귀모델이 되므로 적정 람다값을 구하는 것이 일반화가 잘되는 지점을 찾는 것이며, 이것을 정규화 모델이라고 함.

λ(람다, lambda)란?

- 그 수치가 커지면 회귀식 매개변수의 증가량이 달라진다. 즉 기울기를 조절해 회귀계수를 조정할 수 있는 것이다. **패널티, 규제**의 개념
- 0 ~ ∞까지 나올 수 있으며, 0일 경우 RSS만 최소화 하는 선이 되므로 기존 선형회귀모델과 같게 된다.
- parameter tuning을 통해서 편향과 분산의 균형잡힌 회귀모델을 작성가능하도록 도와준다.

공식에서 계수의 크기의 제곱에 해당하는 페널티가 적용되어, 계수가 작지만 0이 아니도록 보장함.

Linear regression: $\min \sum_i \left(y_i - \beta_0 - \beta_2 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right)^2$

Ridge regression: $\min \sum_i \left(y_i - \beta_0 - \beta_2 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right)^2 + \lambda \sum_{i=1}^{p} \beta_i^2$

Ridge : 손실함수(MSE)에 L2 정규항을 더한 것

Ridge 목적함수의 λΣβΣ 도 커지면 안된다. 람다를 애초에 키우면 베타가 커지는 것을 막아주려고 베타값을 작게 추정할 수 밖에 없는 것이다. 예를 들어 이상치 하나 때문에 베터 계수가 확 커져버리는 그런 문제를 방지한다. 베타가 큰 의미없으면 0에 가깝게 추정되도록 해주는 역할임. 기본 선형회귀는 독립변수들을 스케일링 하지 않아도 성능이 바뀌지 않지만, βΣ 은 그렇지 않아서 변수를 정규화시켜준 후 Ridge regression을 실행해야한다.

특성

- L2-norm 패널티를 가진다. -> λ * (계수들의 제곱합 = L2 Norm의 제곱)
- 릿지의 가중치들은 0에 가까워질 뿐 0이 되지는 않는다. => 0에 가까운 값을 가짐으로써 특정 feature의 중요도를 낮춰주는 역할
- 특성이 많은데 그중 일부분만 중요하다면 라쏘가, 특성의 중요도가 전체적으로 비슷하다면 릿지가 좀 더 괜찮은 모델을 찾아줄 것이다.
- 설명변수가 많아질 때 least square는 유일한 해가 없게되는데, 이런 상황에서 릿지는 bias에서 약간의 손해를 보더라도 variance를 크게 줄여 least square보다 좋은 결과를 가져올 수 있음.

장점

- 다중공선성 문제를 효과적으로 완화시킴. 다중공선성이 있는 경우에도 안정적인 회귀 계수를 추정하여 모델의 예측력을 향상시킬 수 있다.

- L2 penalty를 통해 회귀 계수의 크기를 제한하므로 모델의 안정성이 향상. 데이터의 작은 변화에도 모델이 과도하게 변하지 않아 예측 결과가 더 일반화될 수 있다.

- L2 penalty는 모든 변수를 살려두기 때문에, 모든 변수가 유용한 정보를 가지고 있다고 가정하는 경우에 적합. 따라서 중요하지 않은 변수를 제거하려는 경우보다는 변수들을 모두 고려하는 데 더 적합.

- L2 penalty의 강도를 조절하는 하이퍼파라미터 λ를 조절하여 모델의 성능을 향상시킬 수 있다. 적절한 λ 값을 선택하면 과적합을 방지하고 예측력을 극대화할 수 있다.

단점

- 모든 변수에 대해 균등한 영향을 미치기 때문에, 중요한 변수들을 강조하기 어려울 수 있다. 중요한 변수들을 무시하거나 덜 강조할 수 있어서 모델의 해석이 어려울 수 있다.

- L2 penalty에 의해 회귀 계수들이 축소되므로, 변수들의 실제 영향을 해석하는 것이 어려울 수 있다.

- 릿지 회귀는 변수 선택을 하지 않기 때문에 모델의 성능이 약간 제한될 수 있다. 변수 선택이 중요한 경우에는 다른 회귀 모델을 고려하는 것이 좋을 수 있다.

- 선형 모델이기 때문에 비선형 관계를 잘 처리하지 못할 수 있다. 비선형 문제에 적용하기 위해서는 데이터의 전처리나 다른 회귀 모델을 고려해야 할 수 있다.

R에서 릿지 회귀분석 실시하기 : glmnet 패키지의 glmnet() 함수
R 의 내장 데이터인 state.x77 을 이용해 릿지회귀분석을 실시하여 봅시다. 먼저 state.x77 데이터를 불러와 학습 및 테스트 비율을 8:2 로 나눕니다.

#데이터 불러오기

```
> library(glmnet)
> raw_data <- state.x77
> head(raw_data)
       Population Income Illiteracy Life Exp Murder HS Grad Frost   Area
```

```
Alabama        3615   3624       2.1    69.05   15.1    41.3    20  50708
Alaska          365   6315       1.5    69.31   11.3    66.7   152 566432
Arizona        2212   4530       1.8    70.55    7.8    58.1    15 113417
Arkansas       2110   3378       1.9    70.66   10.1    39.9    65  51945
California    21198   5114       1.1    71.71   10.3    62.6    20 156361
Colorado       2541   4884       0.7    72.06    6.8    63.9   166 103766
```

#학습 및 테스트 데이터 나누기

```
> set.seed(100) #랜덤난수 초기값 고정
> trainIndex <- sample(1:nrow(raw_data),replace=F, size=40)
> train=raw_data[trainIndex, ] #학습용 데이터 80%
> test=raw_data[-trainIndex, ] #테스트용 데이터 20%
```

8 개 변수 중 Life Exp(기대수명)을 종속변수로 두고 나머지 7 개 변수는 예측변수로 나누어 각각 저장합니다. 이 때 예측변수와 종속변수를 scale() 함수를 이용해 표준화시킵니다.

#학습 데이터 예측 및 종속변수 나누기

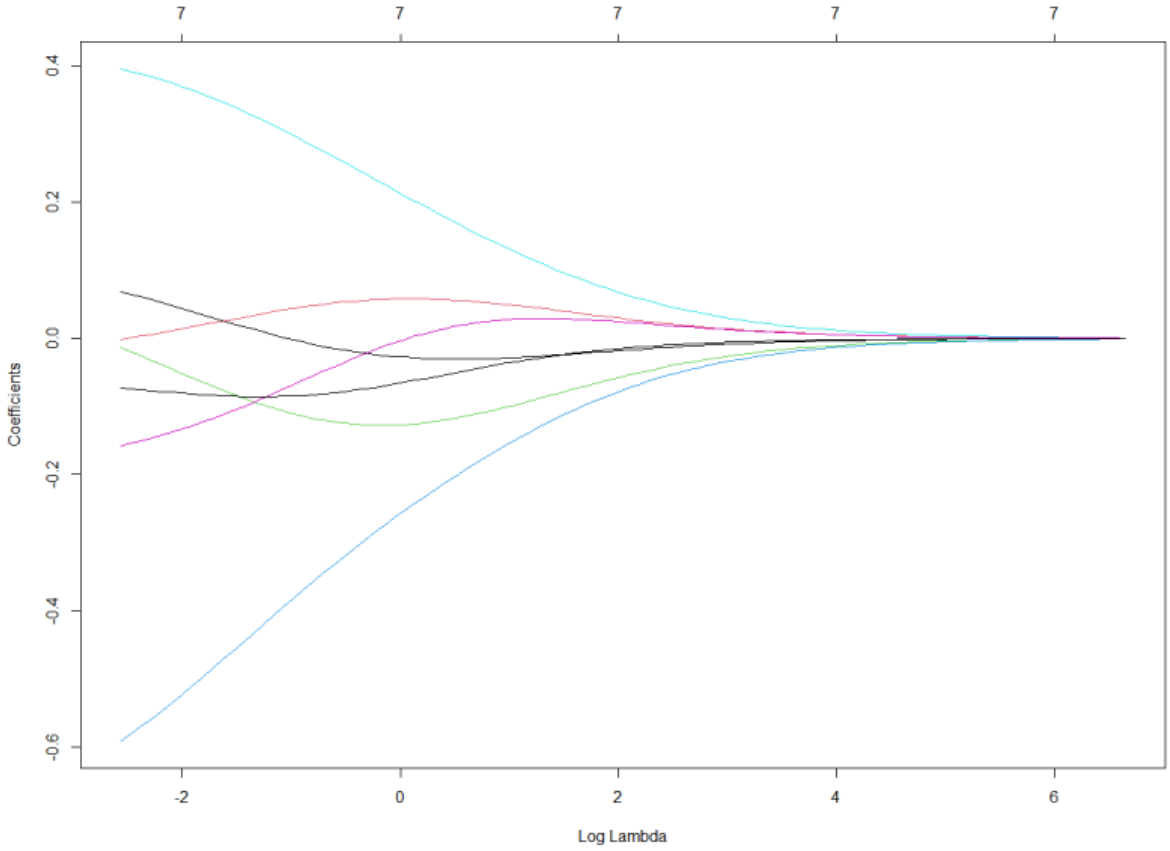
```
> train.x <- scale(train[,-4]) #예측변수 표준화
> train.y <- scale(train[,4]) #종속변수 표준화
>
```

#테스트 데이터 예측 및 종속변수 나누기

```
> test.x <- scale(test[,-4]) #예측변수 표준화
> test.y <- scale(test[,4]) #종속변수 표준화
```

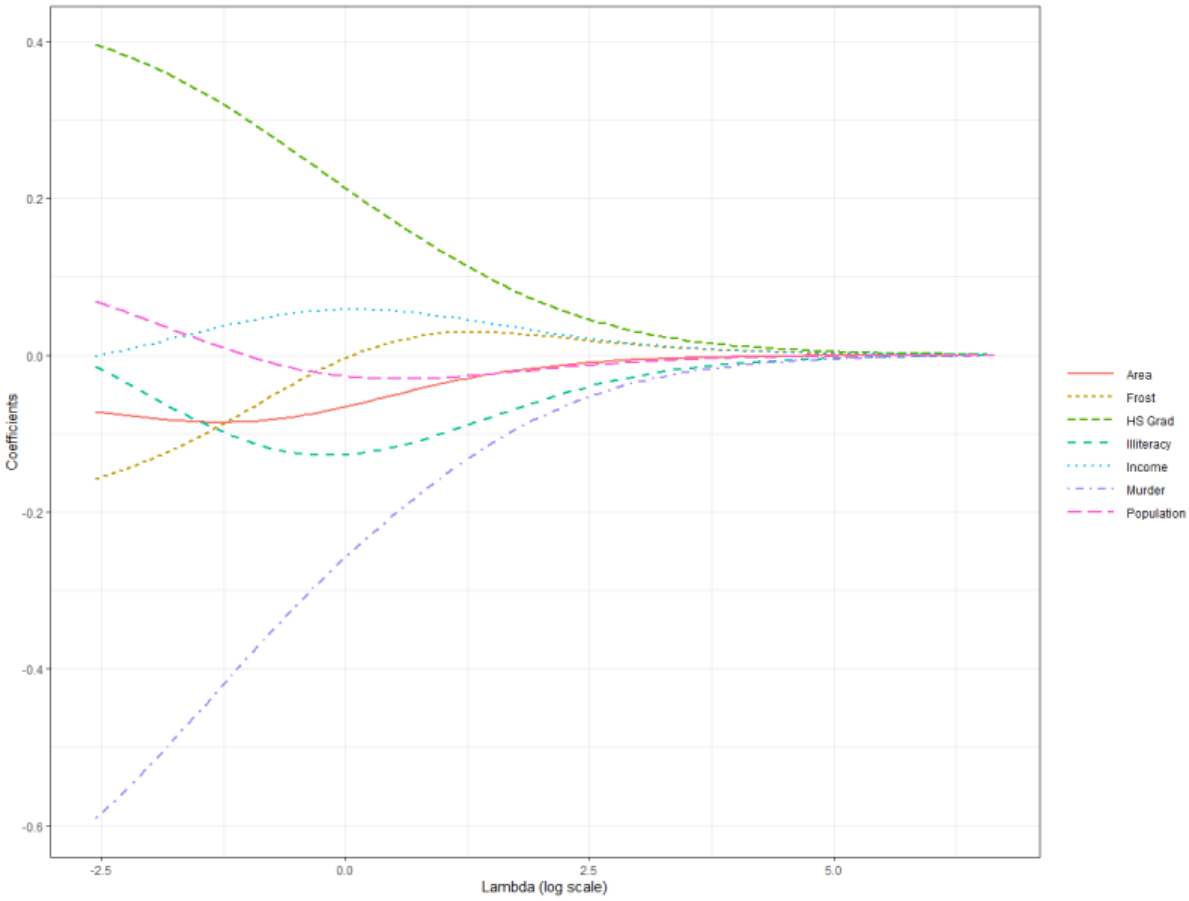
그럼 학습데이터를 이용하여 릿지회귀모델을 만들어 보도록 하겠습니다.

```
> ridge_model <- glmnet(train.x, train.y, alpha=0) #학습시키기
> plot(ridge_model, xvar="lambda") #람다에 따른 회귀계수 확인하기
```



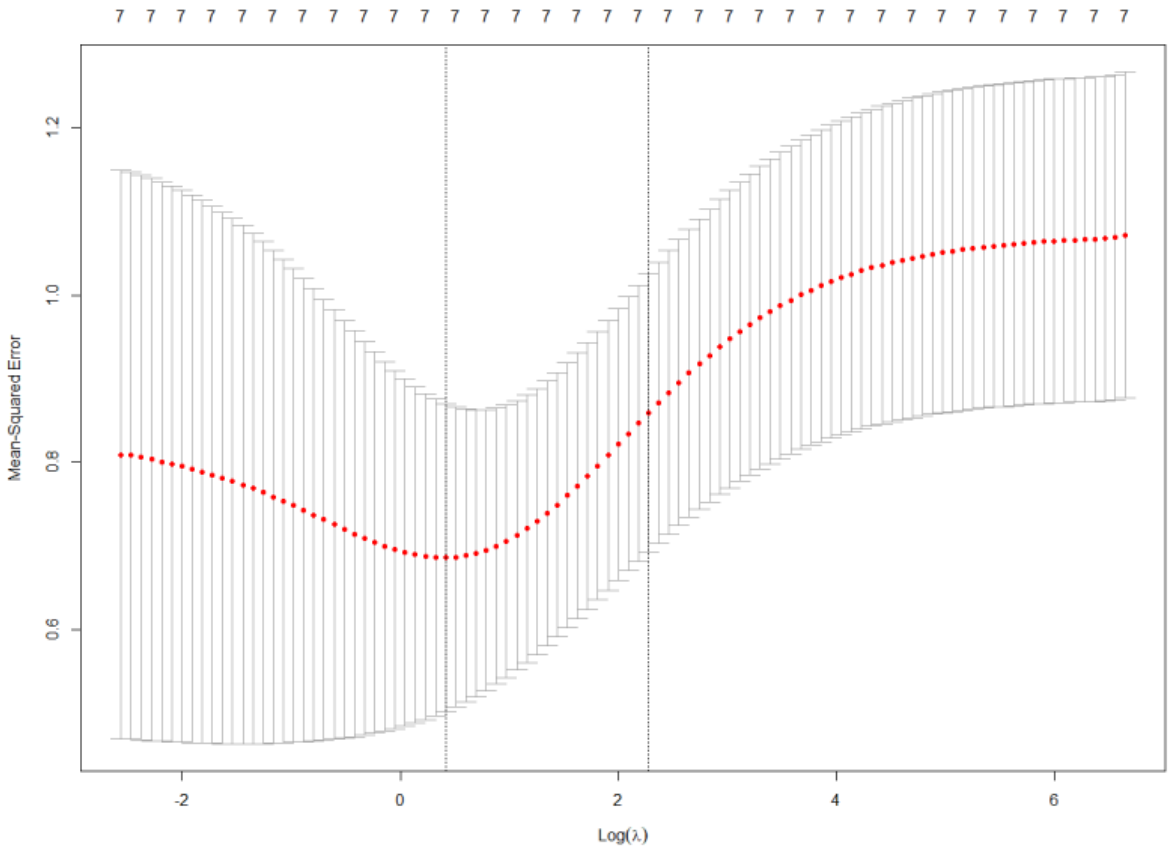
위에서 plot() 함수를 이용하면 λ (조율모수)에 따른 회귀계수의 추정치를 살펴볼 수 있습니다. 전체적으로 λ가 커질수록 회귀계수가 줄어듦을 확인할 수 있습니다. plot() 함수를 이용하면 각 변수의 회귀계수를 라벨링할 수 없어서 아래와 같이 모델결과 데이터를 이용하여 직접 처리해줍니다.

```
#그래프를 그리기위한 데이터 처리
> beta=coef(ridge_model) #회귀계수 추출하기
> tmp <- as.data.frame(as.matrix(beta))
> tmp$coef <- row.names(tmp)
> tmp <- reshape::melt(tmp, id = "coef")
> tmp$variable <- as.numeric(gsub("s", "", tmp$variable))
> tmp$lambda <- log(ridge_model$lambda[tmp$variable+1]) # 람다 추출 및 log10 변환
> tmp$norm <- apply(abs(beta[-1,]), 2, sum)[tmp$variable+1] # compute L1 norm
>
#그래프 그리기
> library(ggplot2)
> ggplot(tmp[tmp$coef != "(Intercept)",], aes(lambda, value, color = coef, linetype
= coef)) +
+   geom_line(size=1) +
+   xlab("Lambda (log scale)") +
+   ylab("Coefficients") +
+   guides(color = guide_legend(title = ""),
+          linetype = guide_legend(title = "")) +
+   theme_bw() +
+   theme(legend.key.width = unit(3,"lines"))
```



다음으로 교차검증방법으로 적절한 λ을 선정합니다. 여기서 사용할 함수는 glmnet 패키지의 cv.glmnet() 함수이며 glmnet() 함수와 같은 인자로 입력합니다. 교차검증방법 특성상 cv.glmnet() 함수를 실행할 때마다 결과가 바뀌기 때문에 set.seed() 함수로 난수를 고정시켜줍니다.

```
#교차검정을 통한 적절한 λ 선정
> set.seed(100) #초기값 고정
> cv.ridge <- cv.glmnet(train.x, train.y, alpha=0) #교차검정 실시
> plot(cv.ridge)
> (best_lambda <- cv.ridge$lambda.min) #오차가 제일 작은 값(최적 λ)
[1] 1.515164
```

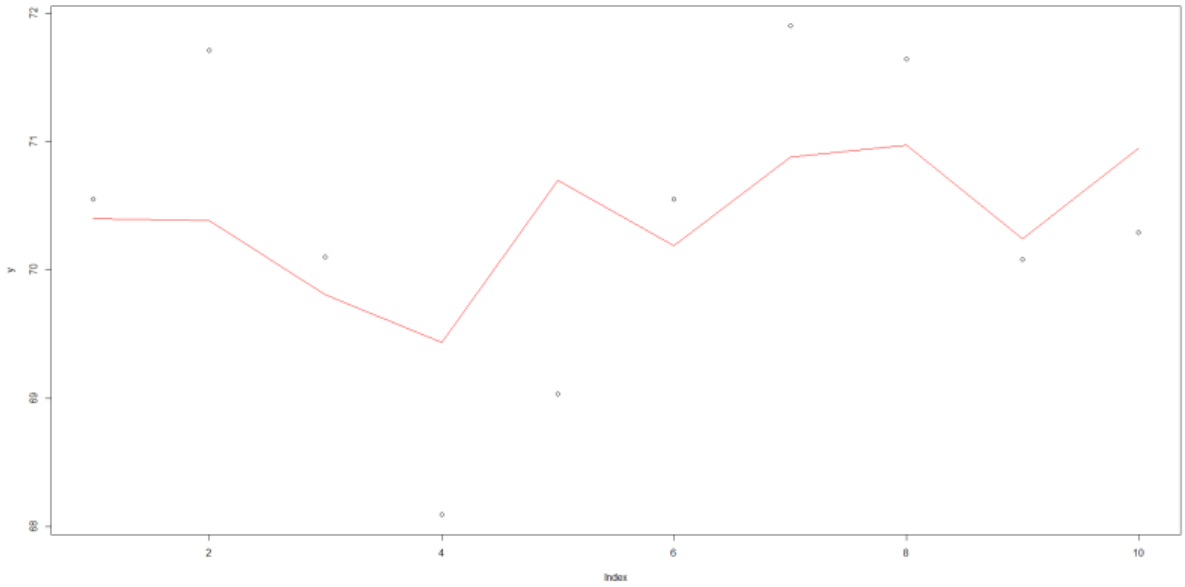


위의 그래프는 log(λ)에 따른 평균 교차타당오차를 나타내며 이를 통해 log(λ)가 0.415 부근에서 오차가 가장 작은 것을 알 수 있습니다. 오차가 가장 작은 λ 을 정확하게 구할려면 cv.glmnet() 함수의 결과의 lambda.min 원소를 통해 구할 수 있으며 이를 적절한 λ 라고 설정할 수 있습니다.
다음으로 적절한 λ 를 다시 학습된 모델에 입력하여 각각 추정된 회귀계수를 추출해보도록 하겠습니다.

```
#적절한 λ에 따른 회귀계수 확인
> (ridge_coef <- coef(ridge_model, s=best_lambda))
8 x 1 sparse Matrix of class "dgCMatrix"
                      1
(Intercept) -4.416119e-15
Population  -3.027289e-02
Income       5.673710e-02
Illiteracy  -1.197011e-01
Murder      -2.113961e-01
HS Grad      1.775025e-01
Frost        1.482564e-02
Area        -5.309407e-02
```

마지막으로 테스트 데이터을 학습된 릿지 회귀모델에 입력하여 종속변수 Life.Exp 를 예측해보도록 합시다. 사용할 함수는 predict() 함수입니다. test.Y 와 예측값(pred_y)은 표준화된 값임으로 역 표준화 후에 그래프로 그려보도록 하겠습니다.

```
> #릿지회귀모형으로 예측하기
> pred_y <- predict(cv.ridge, newx=test.x, s="lambda.min")
 # test Y 역 표준화 후 plot
> plot(test.y * attr(test.y, 'scaled:scale') + attr(test.y, 'scaled:center'), ylab
= "y")
# 예측값 역 표준화 후 plot
> lines(pred_y * attr(test.y, 'scaled:scale') + attr(test.y, 'scaled:center'),
col="red")
```



2. lasso regression

라쏘 회귀(lasso regression)는 규제 회귀 분석의 한 종류로 선형 회귀의 단점을 보완하여 일반화 능력을 향상시키기 위해 만들어진 도구이다.

기존의 선형 회귀에서는 적절한 가중치와 편향을 찾아내는 것이 관건이다. 라쏘는 거기에 덧붙여서 추가 제약 조건 부여한다.
제약: MSE가 최소가 되게 하는 가중치와 편향을 찾는데 동시에 가중치들의 절대값들의 합, 즉 가중치의 절대값들이 최소(기울기가 작아지도록)가 되게 해야 한다. 다시 말해서 가중치(w)의 모든 원소가 0이 되거나 0에 가깝게 되게 해야 한다.
따라서 어떤 특성들은 모델을 만들 때 사용되지 않는다. 어떤 벡터의 요소들의 절대값들의 합은 L1-norm이므로 라쏘는 간단히 말해서 L1-norm 패널티를 가진 선형 회귀 방법이다. 공식5를 보면

$$MSE + penalty$$
$$= MSE + \alpha \cdot L_1\text{-}norm$$
$$= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha\sum_{j=1}^{m}|w_j|$$

(공식1: MSE + penalty)

여기서 m은 가중치의 개수를 의미하고(따라서 특성의 개수도 됨), α는 페널티의 효과를 조절해주는 파라미터이다. α의 값이 커지면 패널티 항의 영향력이 커지고, α의 값이 작아져서 거의 0이 되면 선형 회귀와 같아진다는 것을 알 수 있다. MSE와 penalty 항의 합이 최소가 되게 하는 ω와 b를 찾는 것이 라쏘의 목적이다.

$$\underset{w,\ b}{argmin}\left\{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha\sum_{j=1}^{m}|w_j|\right\}$$

(공식2: MSE + penalty를 최소로 만드는 w, b 찾기)

과대적합을 피하는 일반적인 모델을 만드는 방법
우선 MSE항이 작아질 수록 라벨값들과 예측값들의 차이가 작아지고, L1-norm이 작아질 수록 많은 가중치들이 0이 되거나 0에 가까워진다. 이런 상황에서 α가 크면 많은 가중치들이 0이 되거나 0에 가까워지게 하는 것에 좀 더 큰 비중을 두게 되는 반면, 훈련셋에서의 예측정확도는 상대적으로 덜 중요해진다. 왜냐하면 α가 클 때는 전체값(MSE+penalty)이 L1-norm에 좌우되기 때문이다. L1-norm이 커지면 전체값이 커지므로 전체값을 최소가 되게 하기 위해서는 L1-norm을 작게 만드는 것이 우선순위가 된다. 반면 α가 작으면 훈련셋에서의 예측정확도(MSE)가 작아지게 하는 것에 좀 더 큰 비중을 둔다. 전체값이 MSE에 의해 좌우되기 때문이다. 따라서 α가 너무 작으면 과대적합(복잡도가 너무 큼)이 되고, 너무 크면 과소적합(복잡도가 너무 작음)이 된다.

***L1-norm:**
벡터가 얼마나 큰지를 알려주는 것이 norm인 것이다. 여기서 말하는 크기는 벡터의 차원의 크기가 아니라, 구성요소의 크기(magnitude)를 표현하는 것이다.
L1-norm은 각 원소들의 절댓값의 합으로 표현하며 Lasso 회귀에서 loss값을 규제하는 방법이다. L1-norm은 L2-norm과 비교해서 이상치의 영향을 크게 받지 않는다. L2-norm은 제곱해서 더해주기 때문에 분포에서 벗어난 값이 커져서 전체에 영향이 주는 경우가 많이 발생한다.

$$||x||_1 = \sum_{i=1}^{n} |x_i|$$

(공식3: L1-norm)

Lasso 장점

1. 제약 조건을 통해 일반화된 모형을 찾는다는 것이다.
2. 가중치들이 0이 되게 함으로써 그에 해당하는 특성들을 제외해준다. 결과적으로 모델에서 가장 중요한 특성이 무엇인지 알게 되는 등 모델 해석력이 좋아진다.

Lasso 단점

1. 변수들끼리 correlate한(밀접한 상관관계에 있는) 경우에, Lasso는 단 한개의 변수만 채택하고 다른 변수들의 계수는 0으로 바꿀 것이다. 이렇게 정보가 손실됨에 따라 정확성이 떨어질 수 있다.
2. 변수들의 중요도가 전반적으로 비슷한 경우에 효과적이지 못하다.

R코드

#전체데이터 사용

-Lambda 생략

```
library(glmnet)  #glmnet()
library(Metrics) #mse()

x <- as.matrix(mtcars[, -1])
y <- mtcars[,1]

model <- glmnet(x, y, alpha=1)
length(model$lambda) #79 개의 lambda 값 입력 → 모델 79 개
```

```
79
```

```
coef(model)[, 1]      #1 번 모델의 회귀계수
```

```
(Intercept):  20.090625 cyl:  0 disp:  0 hp:  0 drat:  0 wt:  0 qsec:  0 vs:  0 am:  0 gear:  0 carb:  0
```

```
coef(model)[,79]      #79 번 모델의 회귀계수
```

```
(Intercept):  12.5663417443922 cyl:  -0.09683688139078 disp:  0.0117899480172213 hp:  -0.0204850968270455 drat:
  0.80052488348812 wt:  -3.58989635131779 qsec:  0.796261952928483 vs:  0.29788826768297 am:  2.50193332120098
gear:  0.651973341818651 carb:  -0.243392821968407
```

```
pred <- predict(model, newx=x)
mse(y, pred[, 1])    #1 번 모델의 MSE
mse(y, pred[,79])    #79 번 모델의 MSE
```

```
35.188974609375

4.61091540760244
```

-Lambda 지정

```
library(glmnet)  #glmnet()
library(Metrics) #mse()

x <- as.matrix(mtcars[, -1])
y <- mtcars[,1]

# lambda=1 일 때
model1 <- glmnet(x, y, alpha=1, lambda=1)
coef(model1)
```

```
11 x 1 sparse Matrix of class "dgCMatrix"
                    s0
(Intercept) 35.3137765
cyl         -0.8714512
disp         .
hp          -0.0101174
drat         .
wt          -2.5944368
qsec         .
vs           .
am           .
gear         .
carb         .
```

```
pred1 <- predict(model1, x)
mse(y, pred1)
```

```
6.7288063960854
```

```
# lambda=2 일 때
model2 <- glmnet(x, y, alpha=1, lambda=2)
coef(model2)
```

```
11 x 1 sparse Matrix of class "dgCMatrix"
                    s0
(Intercept) 31.873616663
cyl         -0.799971158
disp         .
hp          -0.002227335
drat         .
wt          -2.022363172
qsec         .
vs           .
am           .
gear         .
carb         .
```

```
# lambda=2 일 때
model2 <- glmnet(x, y, alpha=1, lambda=2)
coef(model2)
```

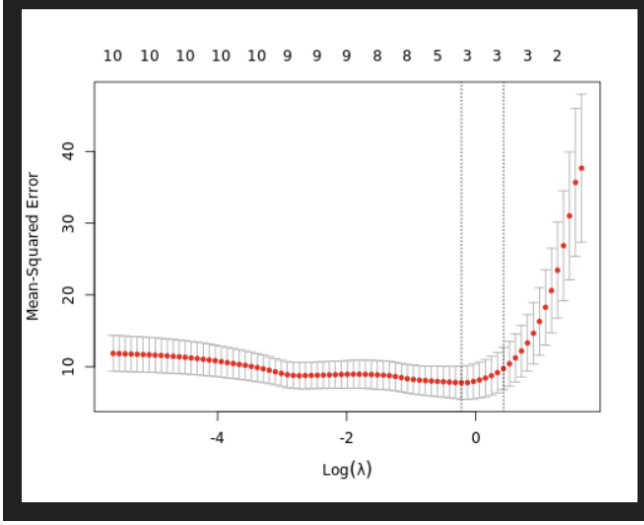
```
10.3562348351776
```

-cv.glmnet()으로 lambda 찾기

```
set.seed(12345) #시드값 지정
library(glmnet) #cv.glmnet()
```

```r
x <- as.matrix(mtcars[, -1])
y <- mtcars[,1]

cv <- cv.glmnet(x, y, alpha=1)
cv$lambda.min #MSE 최소화 lambda = 0.8007036
cv$lambda.1se #MSE 최소화 lambda + 1 표준편차 = 1.535678
plot(cv)
```

```
[1] 0.8007036
[1] 1.535678
```



#데이터 분할

-범위 지정하여 lambda 찾기

```r
set.seed(12345)
library(caret)   #createDataPartition()
library(glmnet)  #glmnet()
library(Metrics) #mse()

df <- mtcars

#데이터 분할 (7:3)
idx <- createDataPartition(df$mpg, list=F, p=0.7)
Train <- df[ idx,]
Test  <- df[-idx,]
Train.x <- as.matrix(Train[, -1])
Test.x  <- as.matrix(Train[, -1])
Train.y <- Train[,1]
Test.y  <- Test [,1]

lambda <- 10^seq(5, -20, by=-.05)
model <- glmnet(Train.x, Train.y, alpha=1, lambda=lambda)
pred <- predict(model, newx=Train.x)
```

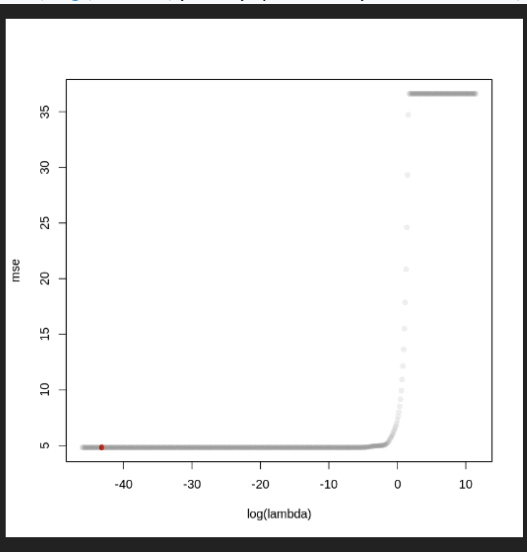
```r
mse <- c()
for( i in 1:length(lambda) )
{
  mse <- append(mse, mse(Train.y, pred[,i]))
}
length(lambda) #확인한 모델(lambda) 수
idx <- which.min(mse) #훈련셋 MSE 최소 모델번호
idx
```

```
501

476
```

```r
mse[idx] #훈련셋 MSE 최소값
lambda[idx] #훈련셋 MSE 최소 lambda
```

```
4.83314365454864

1.77827941003892e-19
```

```r
#훈련셋 MSE 시각화
color = rep("#00000011", length(lambda))
color[idx] = "red"
plot(log(lambda), mse, pch = 16, col = color)
```



3. XGBoost

XGBoost는 (Extreme Gradient Boosting)의 약어로
기존 Gradient Boosting 알고리즘에 과적합 방지를 위한 기법이 추가된 지도학습 알고리즘
information gain 기반 importance 산출 기능을 통해 효과적인 모델학습 성능을 보여줌.

*** Gradient Boosting이란?**
Gradient(또는 잔차(Residual))를 이용하여 이전 모형의 약점을 보완하는 새로운 모형을 순차적으로 적합한 뒤 이들을 선형 결합하여 얻어진 모형을 생성하는 지도학습 알고리즘.

XGBoost가 CART(Classification And Regression Trees)를 기반으로 만들어진 모델이며
이름 그대로 분류 및 회귀 나무'들'로 트리 기반 앙상블 모델을 의미. 앙상블 모델은 다수의 학습방법(bagging, boosting, staking...)을 이용하여 결론을 내리는 방법으로, CART는 이 '다수의 학습방법'을 여러가지 의사결정나무(Decision Tree)로 정의
쉽게 말해서 여러가지 Tree를 더해서 학습시키는 모델이며 아래 수식으로 표현가능.

$$Y' = a * tree\ A + b * tree\ B + c * tree\ C + \cdots ..$$

$$where\ a, b, c \ldots . = weights\ for\ each\ tree$$

Y'는 타겟(Y)에 대한 예측값을, a,b,c…는 각각 트리 A,B,C…에서 나온 가중치들을 의미함. 이 개념을 XGBoost의 Gradient Boosting Tree로 가져가면 다시 밑과 같이 표현됨.

$$y'_i = \sum_{k=1}^{K} f_k(x_i)\ , f_k\ \in F$$

$$\text{Obj} = \sum_{i=1}^{n} l\left(y_i, y'_i\right) + \sum_{k=1}^{K} \Omega(f_k)$$

$$* where\ y'_i = \text{predict score corresponding to } x_i,$$

$$f_k = k\ th\ decision\ tree\ in\ function\ space\ F,$$

$$l = loss\ function,$$

$$\Omega = regularization\ term(= complexity\ of\ Trees)$$

f함수는 알고리즘에서 생성해낸 의사결정나무 모델들을 의미하고 l은 loss function을, regularization term은 나무들이 얼마나 복잡할지에 대해 정의하는 파라미터.
의사결정 나무 모델을 여러 개 학습시켜서 예측값이 더해진 예측 점수들을 이용해 결론을 내림으로서 과적합이나 기존 모델이 잘 설명하지 못하는 취약부분에 대한 보완하게 됨.

이후 여러가지 모델을 생성할 때 무슨 기준으로 만들어야 하는지, 위에 말한 **regularization term(모델 복잡도)**은 어떻게 정해야 하는지, gradient boosting의 꽃인 **learning rate**은 어떻게 설정해야 할지 등의 문제를 해결하기 위해 유저가 설정해주는 **'하이퍼 파라미터'**가 존재함.
XGBoost에서는 하이퍼 파라미터를 얼마나 잘 조정해주느냐가 성능을 끌어올리는데 중요한 역할을 하는데 아래에서 주요한 하이퍼 파라미터를 수식과 함께 소개하겠음.

~~~~~~

-eta : learning rate, default = 0.3, [0, 1]
eta는 'Step size shrinkage'로, 학습 단계별로 가중치를 얼마나 적용할 지 결정하는 숫자임.가중치이므로 0~1 사이의 값을 가지며, 낮을수록 more conservative, 즉 보수적인 모델이 됨. (다음 단계의 결과물을 적게 반영하기 때문)

-gamma : min split loss, default = 0, [0, ∞]
Gamma를 이해하기 위해서는 먼저 information gain이라는 것을 이해해야 하는데,
Information gain은 의사결정나무가 가지를 칠 때, 즉 새로운 변수를 기준으로 데이터를 분류할 때 타겟 변수에 대해 얼마나 설명할 수 있는지를 측정하는 기준.

$$\text{Gain} = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma$$

XGBoost에서의 Information Gain 수식
여기서 감마는 맨 끝의 숫자로, 계산한 information gain에 페널티를 부여하는 숫자이며 이 값이 커질수록 의사결정나무들은 가지를 잘 만들려 하지 않게 되고 보수적인 모델이 됨.

-max_depth : default = 6, [0, ∞]
max_depth는 말 그대로 의사결정나무의 깊이의 한도. 기본 값은 6으로 커질수록 더 복잡한 모델이 생성되며, 이는 overfitting의 문제를 일으킬 수도 있음. 이 값은 어디까지나 한계치이기 때문에 무작정 늘린다고 모든 나무들이 많은 가지를 생성해내지는 않음을 유의.

-subsample : default = 1, (0, 1]
training 데이터셋에서 subset을 만들지 전부를 사용할지를 정하는 파라미터. 매번 나무를 만들 때(=iteration) 적용하며 overfitting 문제를 방지하려고 사용.

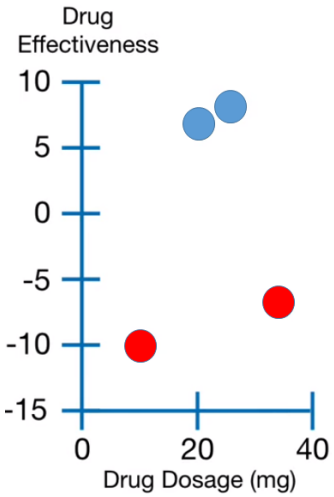
-colsample_bytree : default = 1, (0, 1]
나무를 만들 때 칼럼, 즉 변수를 샘플링해서 쓸지에 대한 파라미터. 나무를 만들기 전 한 번 샘플링 진행.

-scale_pos_weight : default = 1, (0, 1]
분류 모델에서 사용하는 가중치 파라미터로, 극단적으로 적은 타겟값이 존재하는 문제에서 유용.
~~~~~~

하이퍼 파라미터는 대부분 값이 정해진 규칙이 없기에 RandomSearch나 GridSearch로 최적값을 찾게되며, 어떤 방법이든 heuristic한 방법들이기 때문에 많은 연산을 필요로 하며, 따라서 무작정 넓은 값을 찾기보다는 현 데이터 상황에 맞는 중요한 파라미터들을 선별하고 그 의미를 알고 있는 상태에서 찾는 것이 중요.

2. 학습 알고리즘

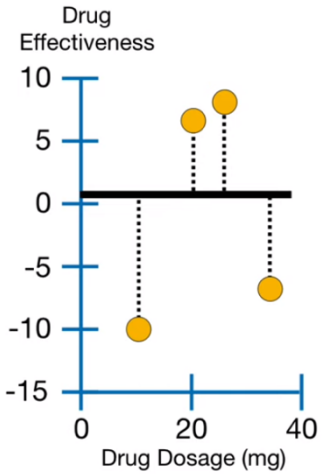
*해당 예시는 Regression을 예로 들며 Classification 도 거의 비슷한 방법으로 학습됨



좌측과 같은 데이터가 있다고 가정하자. X축은 복용량이고, Y축은 효과이다.

즉, 빨간색은 별로 효과를 보지 못한 경우, 파란색은 효과가 나타난 경우이다.

해당 데이터셋을 통해 복용량에 따른 효과 정도를 예측하는xgBoost 모델을 만들어보자.



B. 초기 예측값은 따로 설정하지 않으면 0.5로 초기화된다. 검은색 진한 줄로 표시되어 있다.

(즉, 복용량이 어떻든 효과가 0.5라고 결론 내는 하나의 decision tree이다.)

이 단일 예측을 트리 구조처럼 Gain을 가장 극대화하는 방법으로 여러 예측값으로 나눠야 한다.

Gain을 계산 법은 다음과 같다.

$$Gain = (New\ Similarity\ Score) - (Old\ Similarity\ Score)$$

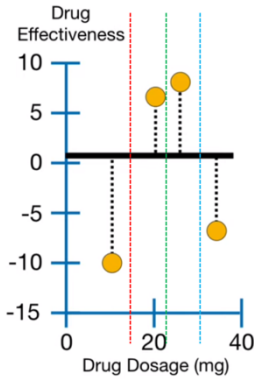
$$\text{Similarity Score} = \frac{\text{Sum of Residuals, Squared}}{\text{Number of Residuals} + \lambda}$$

Smilarity Score를 계산하는 방법은 다음과 같다.

이 케이스에서 Old Similarity Score는 다음과 같이 계산된다.

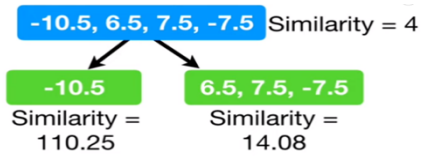
$$\text{Similarity Score} = \frac{(-10.5 + 6.5 + 7.5 + -7.5)^2}{\text{Number of Residuals} + 0}$$

이제 New Simlarity Score를 계산해보자.

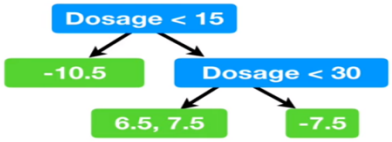


트리의 분기 조건을 각각 빨간 줄, 초록 줄, 파란 줄로 설정하고 나누어 보았을 때의 New Similarity Score를 계산한다.

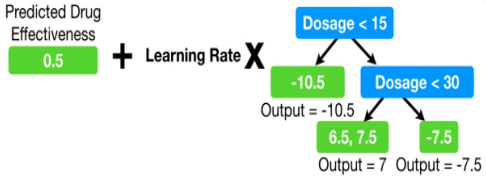
총 유사도(Similarity Score)는 모든 리프 노드의 유사도의 합이다.
예를 들어, 이 예제에서 빨간 줄로 나눴을 때의 총유사도는 다음과 같이 110.25 + 14.08이다.



그리고 Gain값은 (110.25 +14.08) - 4 가 되겠다.
빨간, 초록, 파랑으로 각각 나누어보고 **Gain 값이 가장 높은 쪽**으로 분기한다.
여기에서는 빨강으로 나누었을 때 Gain 값이 가장 높으므로 아래와 같이 분기한다.



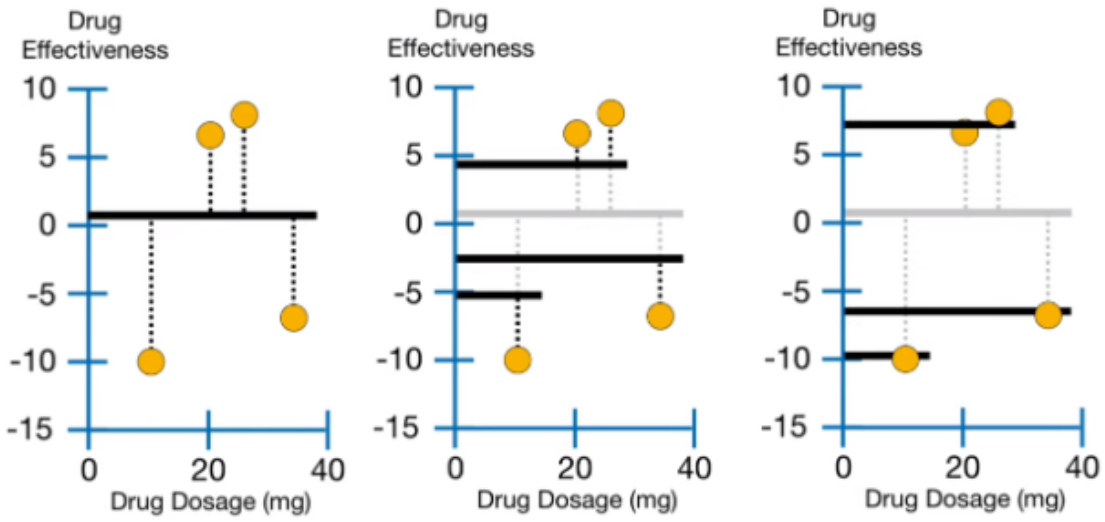
빨간색을 기준으로 분기한 **Tree를 만들고 리프 노드(Leaf node)에 대해서 위 작업을 재귀적으로 반복**한다.
만약 **모든 조건에서 Gain값이 음수**일 경우 분기를 중지한다. 이 예제에서는 좌측과 같이 완성된다.



이후 각각의 예측값을 위의 식으로 변경한 트리를 완성한다. 학습률(Learning Rate)은 모델에 따라 다르게 지정하며 라이브러리에서는 eta로 표기된다.

지금까지의 과정을 통해 새로 만들어진 Tree에 대해서 반복한다.
N회 반복하거나 M회 이상 validation 결과가 개선되지 않을 때 종료한다. (N과 M은 분석자가 직접 지정.)

.



순서대로 첫번째, N번째, 마지막 Tree의 예측값 표시

결론: 이 예제에서는 순차적으로 위와 같이 변화한다.

이렇게 만들어진 모든 트리마다 각각의 weight를 갖고 부스팅 방식으로 최종 output을 결정한다.

3. R에서의 XGBoost를 이용한 회귀분석

#필요한 패키지 설치

```
install.packages("xgboost")
install.packages("data.table")
library(xgboost)
library(data.table)
```

#data.table 패키지는 XGBoost와 함께 데이터를 효율적으로 처리할 때 사용됨.

```
# 임의의 데이터 생성

set.seed(123)
n <- 1000
X <- matrix(rnorm(n * 10), n, 10)
y <- X %*% rnorm(10) + rnorm(n)

# 학습용 데이터와 테스트용 데이터 분리

train.index <- sample(1:n, n*0.8)
X_train <- X[train.index,]
y_train <- y[train.index]
X_test <- X[-train.index,]
y_test <- y[-train.index]
```

```
# 데이터를 xgb.DMatrix 형식으로 변환

dtrain <- xgb.DMatrix(data = X_train, label = y_train)
dtest <- xgb.DMatrix(data = X_test, label = y_test)

#XGBoost는 데이터를 xgb.DMatrix 형식으로 사용. 이 형식은 XGBoost의 내부적으로 최적화된 데이터 구조임.


# XGBoost 모델 매개변수 설정

params <- list(
  booster = "gbtree",
  objective = "reg:squarederror",
  eta = 0.1,
  gamma = 1.0,
  max_depth = 6,
  subsample = 0.5,
  colsample_bytree = 0.9,
  eval_metric = "rmse"
)

#GridSearch나 RandomSearch와 같은 방법으로 최적의 매개변수 값을 찾는 것이 좋음.


# 모델 학습

num_rounds <- 100
model <- xgb.train(
  params = params,
  data = dtrain,
  nrounds = num_rounds,
  watchlist = list(train=dtrain, test=dtest),
  early_stopping_rounds = 10,
  print_every_n = 10
)

# 예측값 얻기

y_pred <- predict(model, dtest)

# RMSE 계산

rmse <- sqrt(mean((y_test - y_pred)^2))
print(rmse)

#RMSE 외에도 MAE(Mean Absolute Error)나 MAPE(Mean Absolute Percentage Error)와 같은 다른 성능 평가 지표를 사용할 수도 있으며 사용하는 지표는 문제의 특성과 요구사항에 따라 달라짐.
```

5. Random Forest

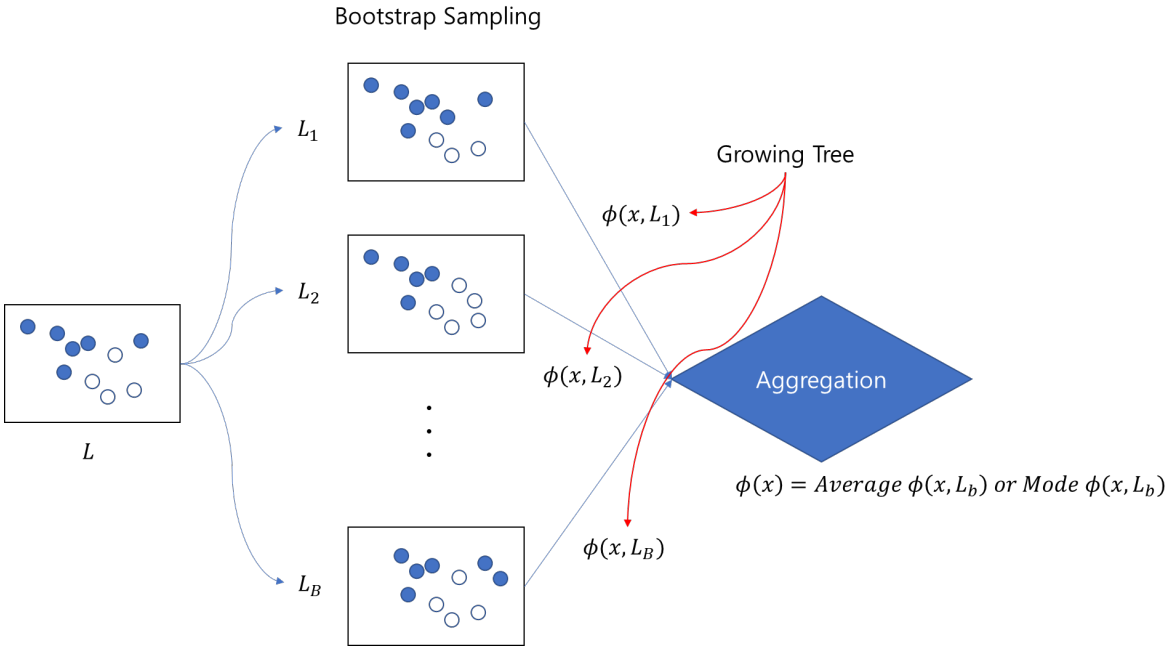
- 정의 -

랜덤 포레스트(Random Forest)는 (1) 기존 배깅의 이점을 살리고 (2) 변수를 랜덤으로 선택하는 과정을 추가함으로써 개별 나무들의 상관성을 줄여서 예측력을 향상한 앙상블 모형이다.

위 정의를 하나하나 살펴보자.

a. 랜덤 포레스트는 배깅을 사용하는 앙상블 모형이다.

랜덤 포레스트는 기본적으로 배깅을 사용한다. 따라서 배깅의 장점인 편의(Bias)를 유지하면서 분산을 낮추는 효과를 랜덤 포레스트 또한 그대로 이어받는다. 아래 그림은 랜덤 포레스트가 배깅을 사용하는 과정을 나타낸 것이다.



랜덤 포레스트에서 배깅 과정

먼저 학습데이터 L=(xi,yi),i=1,⋯,n 이 있다. xi 는 설명변수이고 yi 는 반응 변수이다. 이때 yi 는 수치적인 의미를 갖거나 범주를 나타내는 숫자이다. 이제 L 로부터 B 개의 붓스트랩 샘플 데이터 셋 L1,⋯,LB 을 만들어낸다(Lb 도 L 과 같이 n 개의 데이터를 갖고 있다). 다음으로 각 붓스트랩 샘플 데이터 Lb 를 이용하여 의사결정나무(개별 나무) φ(x, Lb) 를 학습시킨다. 마지막으로 주어진 입력데이터(설명변수) x 를 B 개의 개별 나무 φ(x,Lb) 에 넣어서 얻은 출력값들을 집계하여 예측값을 얻게 된다.

지금까지만 보면 랜덤 포레스트와 배깅이 다를 게 없어 보인다. 하지만 다음에서 소개할 내용이 랜덤 포레스트와 기존 배깅의 큰 차이점이며 이것이 바로 랜덤 포레스트의 진수라 할 수 있다.

b. 랜덤 포레스트는 변수를 랜덤으로 선택하는 과정을 통해 개별 나무들의 상관성을 줄여 예측력을 향상한다.

랜덤 포레스트는 붓스트랩 샘플 데이터 셋을 이용하여 여러 개의 개별 나무들을 만든다고 했다. Breiman 은 그의 논문 "Random Forests"에서 개별 나무 간의 상관성이 작아지면 랜덤 포레스트의 일반화 오류(Generalized Error)가 작아진다는 것을 증명했다. 즉, 개별 나무들의 상관성을 줄이면 랜덤 포레스트의 예측력이 좋아진다는 뜻이다.

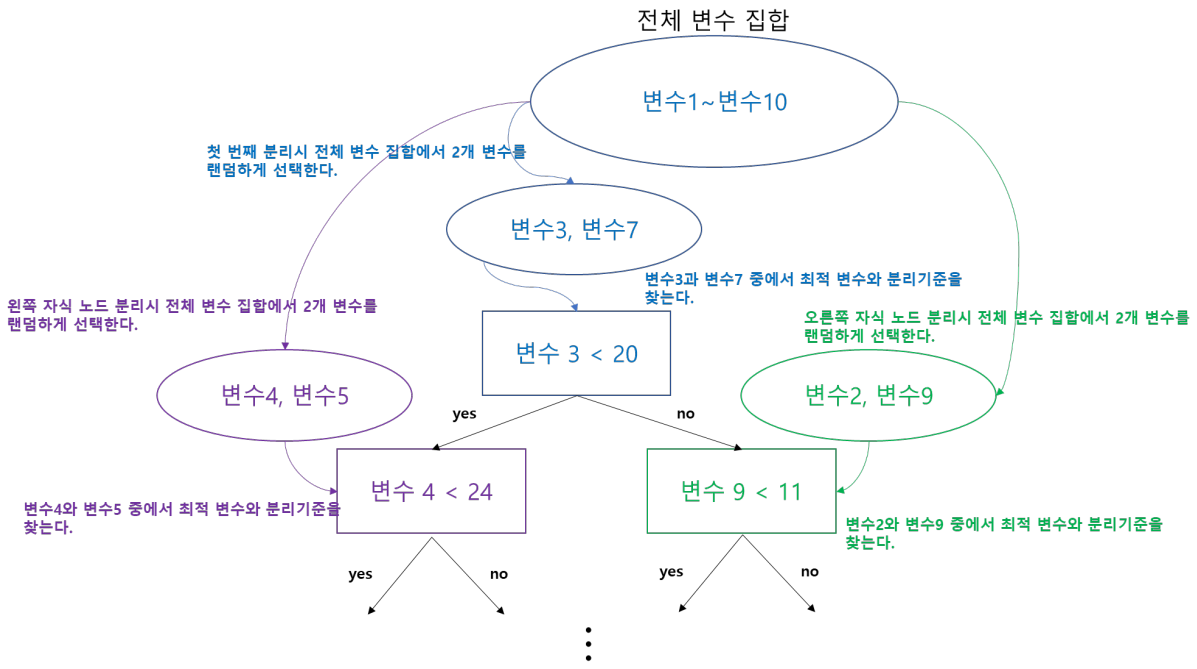
이제 개별 나무들의 상관성이 작아지면 예측력이 향상된다는 것을 알았다(위대한 통계학자 Breiman 이 그렇다고 했으니 그냥 믿으면 된다).

그렇다면 개별 나무들의 상관성을 어떻게 낮출 수 있을까?

→ 개별 나무를 분리할 변수의 후보를 랜덤으로 선택하는 것이다.

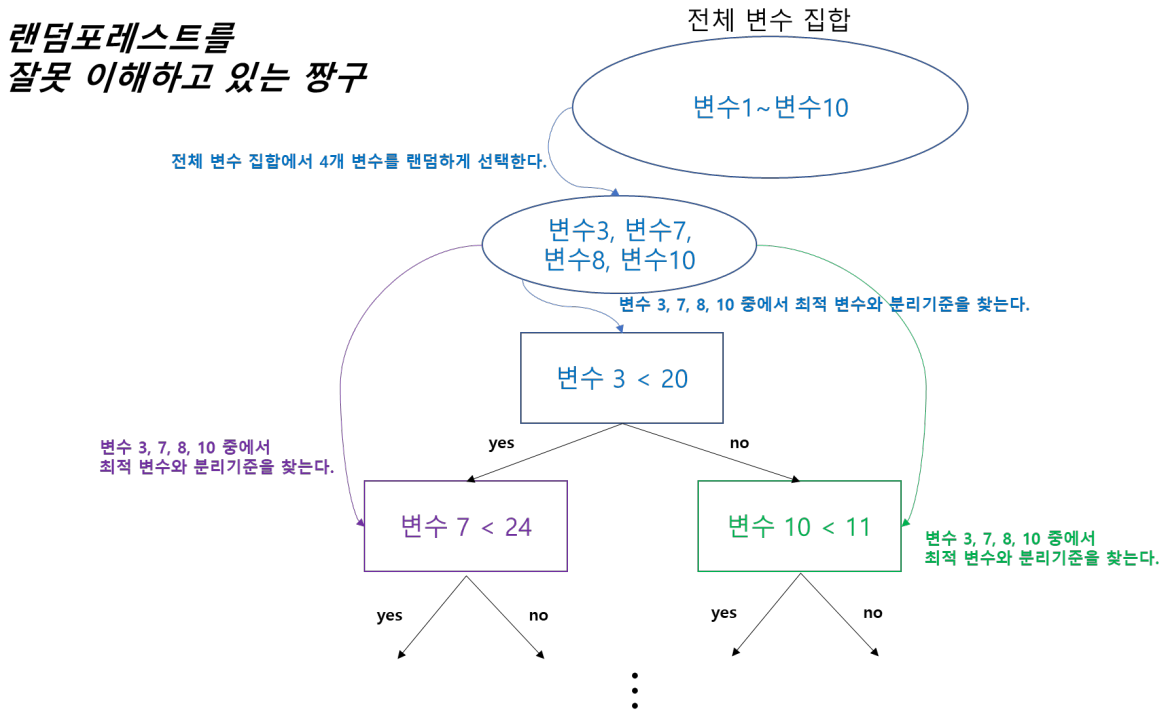
여기서 헷갈리지 말아야 할 것은 각 분리마다 변수 후보를 랜덤하게 선택하는 것이다.

아래 그림은 전체 10 개의 변수 중에서 분리할 때마다 2 개씩 랜덤하게 변수를 선택하고 그중에서 최적 변수와 분리기준을 선택하는 과정이다. 이것이 랜덤 포레스트에서 개별 나무의 성장 과정이다.



랜덤 포레스트에서 개별 트리의 랜덤 변수 선택 과정
변수 후보를 한 번만 랜덤하게 선택해 놓고 그 변수들만 가지고 분리를 하는 것이 아니라는 것을 알아야 한다(이거 은근히 헷갈리는 분들이 많다고 한다).

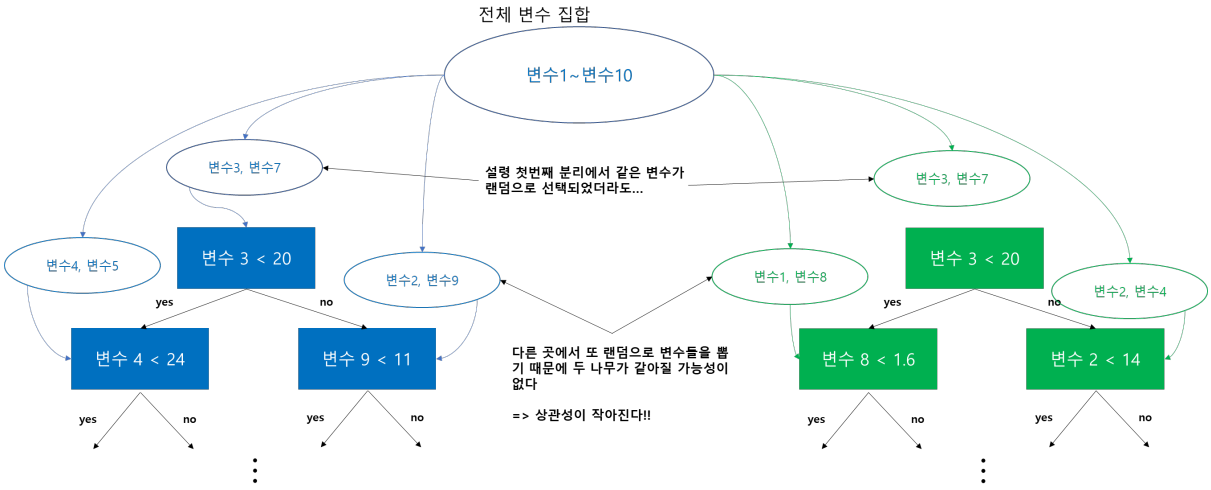
즉, 아래와 같은 과정과 헷갈리지 않도록 하자.



랜덤 포레스트(Random Forest)를 잘못이해하고 있는 사람들이 많다.
이제 랜덤 포레스트에서 개별 트리에서 분리 때마다 변수 후보를 랜덤하게 선택한다는 것을 이해했다. 이러한 과정이 개별 나무 간의 상관성을 어떻게 줄여준다는 것일까?

→ 아래 그림 속에 답이 있다.

아래 그림은 2 개의 나무로 이루어진 랜덤 포레스트 모형을 나타낸 것이다. 개별 나무 내에서 분리할 때마다 랜덤으로 변수 후보를 선택한다면 개별 나무들끼리 같아질 가능성이 거의 없어진다. 이는 곧 개별 나무간 상관성이 작아짐을 뜻한다.



랜덤 포레스트에서는 분리 때마다 랜덤으로 후보 변수들을 뽑기 때문에 트리간 상관성이 작다.

종합하면...

랜덤 포레스트는 개별 나무를 성장할 때 분리마다 랜덤으로 변수 후보를 선택하고 이를 통해 2) 개별 나무 간 상관성을 줄여준다. 그리고 Breiman 은 개별 나무간 상관성이 작은 경우 랜덤 포레스트의 예측력이 올라간다는 것을 수식으로 증명한 것이다.

---

2. 랜덤 포레스트(Random Forest) 알고리즘

랜덤 포레스트 모형 학습 알고리즘은 다음과 같다.

Algorithm

1. 학습 데이터 L, 붓스트랩 샘플들의 개수 B, 랜덤으로 뽑을 변수의 수 F 를 설정한다.
(참고로 총 변수 개수를 p 라 할 때 F=p 이면 랜덤 포레스트는 배깅과 같아진다.)

2. b=1,⋯,B=1,⋯,에 대하여 (a)~(c)를 반복한다.
(a) 붓스트랩 샘플 Lb 를 생성한다. 그리고 OOB 데이터 Ob=L−Lb 보관해둔다.

(b) Lb 를 이용하여 의사결정나무 φ(x,Lb) 를 성장시킨다. 이때 매 분리마다 F 개의 랜덤 변수를 선택한다.
-> 물론 이 과정에서 φ 의 최대 깊이, 끝마디 수, 끝마디 샘플 수 등을 설정해야 할 것이다.

(c) Ob 를 이용하여 성능 지표 eb 를 계산한다.

3. 집계(Aggregation)하여 최종 랜덤 포레스트 모형 φ(x) 를 만든다. 회귀 문제인 경우

$$\phi(x) = \frac{1}{B}\sum_{b=1}^{B}\phi(x, L_b)$$

분류 문제인 경우

$$\phi_B(x) = \text{Mode}\ \ \phi(x, L_b)$$

가 된다. 그리고 랜덤 포레스트 모형의 성능 지표

$$\bar{e} = \frac{1}{B}\sum_{b=1}^{B} e_b$$

를 계산한다.

3. 장단점 및 고려사항

- 장점 -

a. 출력 변수와 입력 변수간 복잡한 관계를 모델링할 수 있다.

b. 예측력이 좋다.

개별 나무 간 상관성이 적다면 우수한 예측력을 가진 랜덤 포레스트를 만들 수 있다.

c. 이상치에 강하다(Robust).

랜덤 포레스트는 이상치에 강하다. 왜냐하면 랜덤 포레스트도 결국 의사결정나무로 이루어져있고 의사결정나무 자체가 입력 변수 이상치에 영향을 덜 받기 때문이다.

- 단점 -

a. 모형의 해석이 어렵다.

랜덤 포레스트도 앙상블 기법으로써 앙상블의 고질적인 문제는 모형의 해석이라는 것이다. 즉, 입력 변수에 따른 출력 변수 변화를 해석하기가 어렵다. Breiman 은 이러한 점을 해결하기 위해 출력 변수의 예측에 영향을 주는 입력 변수들의 중요도를 랜덤 포레스트를 통해 계산하는 방법을 제안하기도 했다.

b. 다른 앙상블처럼 유연하지 않다.

다른 앙상블 기법(예: 부스팅, 배깅 등)에서는 기본 학습기(Base Learner)를 의사결정나무뿐 아니라 다른 여러 가지 모형도 적용할 수 있지만 랜덤 포레스트는 의사결정나무를 겨냥해 만든 것이므로 나무 외에 다른 예측 모형을 적용할 수 없다.

c. 계산량이 많고 학습에 소요되는 시간이 하나의 의사결정나무에 비해 많다.

- 고려 사항-

랜덤 포레스트에서는 붓스트랩 샘플 수 B, 랜덤으로 선택할 변수 개수 F 를 고려해야 한다. 총변수의 개수를 p 라 하자. 경험상으로 B=10 p=10로 시작하는 것이 좋은 시작점이 될 수 있다고 한다. 또한 회귀 문제인 경우 $F \approx p/3$ 분류 문제인 경우 $F \approx \sqrt{p}$ 를 기본 설정값으로 권장하고 있다. 하지만 B 와 F 모두 교차검증을 통한 조절(Tuning) 작업으로 최적값을 계산할 수도 있을 것이다.

Iris 데이터셋을 통한 Random Forest 실습

1. 아이리스(iris)데이터 살펴보기

랜덤 포레스트 알고리즘을 적용하기 위해서 R에서 기본적으로 제공하는 iris를 사용하겠다. iris데이터는 아래와 같이 "Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width", "Species" 5개 컬럼으로 되어 있으며 Species 컬럼에는 아이리스 종류 "setosa", "versicolor, "virginica" 등이 3종류가 있다.

```
#iris 데이터 살펴보기
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
> levels(iris$Species)
    [1] "setosa"     "versicolor" "virginica"
```

랜덤포레스트 알고리즘을 적용하는 목적은 위의 "Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width" 수치형 4개를 독립변수로 이용하여 아이리스의 종류(Species)를 예측하는 것이 되겠다.

- **randomForest 패키지의 randomForest() 함수로 랜덤 포레스트 모델 학습 시키기**

R에서 랜덤 포레스트알고리즘을 학습시킬 수 있는 함수는 randomForest 패키지의 randomForest() 함수다. randomForest() 함수의 입력인자는 두 개의 방법으로 다르게 입력할 수 있다(오버로딩)

```
> library(randomForest)
> train <- sample(1:150, 100) #무작위로 100 개 추출 (학습데이터)
    > rf <- randomForest(Species ~ ., nodesize = 3, mtry = 2, ntree = 15, d
    ata=iris)
```

```
#학습 데이터의 오분류표
> rf$confusion
           setosa versicolor virginica class.error
setosa         50          0         0        0.00
versicolor      0         46         4        0.08
virginica       0          6        44        0.12
```

위의 테이블에서 행이 실제 iris Species 이며 열(setosa ~ virginica)이 랜덤 포레스트 모델로 예측된 분류이다. 행렬의 대각원소가 모델에 의해 정분류된 Case 이며 이외는 오분류라고 판단하고 되겠다. 위의 행렬을 보면 10 개를 제외하고 모두 정분류됨을 확인할 수 있다. 참고로 맨마지막열은 각 분류의 오분류율이다.

- 학습된 랜덤 포레스트 모델으로 Test셋 예측하기

rf라는 변수에 학습된 랜덤 포레스트 모델정보가 저장되어 있기 때문에 이를 이용해 Test 셋에 X인자를 넣어 어떻게 예측하는지 확인해보겠다.
이때, predict() 함수를 이용하여 랜덤 포레스트 모델 정보를 넣고 Test할 데이터 셋을 넣게 되겠다.

```
#Test셋 예측하기
> predict(rf, newdata = iris[-train, ])
         3          4          5          9         10         20         22         23
    setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
        36         37         39         41         43         45         46         52
    setosa     setosa     setosa     setosa     setosa     setosa     setosa versicolor
        62         63         64         66         68         70         71         74
versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicolor
        77         81         83         86         88         92         94         97
versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicolor
       106        111        112        113        116        118        120        123
 virginica  virginica  virginica  virginica  virginica  virginica versicolor  virginica
       134        135        139        143        144
 virginica  virginica  virginica  virginica  virginica
Levels: setosa versicolor virginica
```

위의 코드에서 "predict(rf, newdata = iris[-train,])"은 랜덤 포레스트 모델을 통해 예측된 class(Iris 예측분류)가 되겠다. 전체적으로 랜덤 포레스트 모델의 성능이 얼마나 되는지에 대해 정오분류표을 작성해봤다.

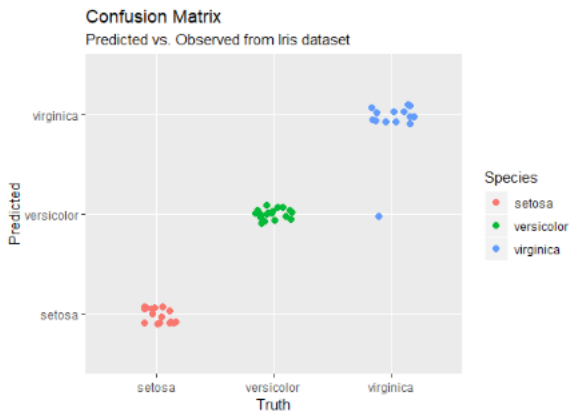
```
> #정오분류표(confusion matrix) 작성
> (tt <- table(iris$Species[-train], predict(rf, iris[-train,])))

           setosa versicolor virginica
  setosa       16          0         0
  versicolor    0         20         0
  virginica     0          1        13
```

위의 테이블에서 행이 실제 iris Species이며 열이 랜덤 포레스트 모델로 예측된 분류이다. 행렬의 대각원소가 모델에 의해 정분류된 Case이며 이외는 오분류라고 판단하고 되겠습니다. 위의 행렬을 보면 1개를 제외하고 모두 정분류됨을 확인할 수 있다.

```
> #정분류율 및 오불류율 계산
> sum(tt[row(tt) == col(tt)])/sum(tt) #정분류율
[1] 0.98
> 1-sum(tt[row(tt) == col(tt)])/sum(tt) #오분류율
  [1] 0.02
```

학습된 랜덤 포레스트 모델은 정분류율은 98%이며 오분류율은 반대로 2%임을 확인할 수 있다.
참고로 정오분류표를 간단하게 그래프로 나타내면 다음과 같다.

```
#정오분류표 그래프화
> library(ggplot2)
> test <- iris[-train,]
> test$pred <- predict(rf, iris[-train,]) #예측된 분류 입력하기
> ggplot(test, aes(Species, pred, color = Species)) +
+    geom_jitter(width = 0.2, height = 0.1, size=2) +
+    labs(title="Confusion Matrix",
+            subtitle="Predicted vs. Observed from Iris dataset",
+            y="Predicted",
+            x="Truth")
```



요약

- **랜덤 포레스트(Random forest)**는 분류, 회귀 분석 등에 사용되는 앙상블 학습 방법의 일종으로, 훈련 과정에서 구성한 다수의 결정 트리로부터 부류(분류) 또는 평균 예측치(회귀 분석)를 출력함으로써 동작하는 알고리즘
- 랜덤 포레스트와 배깅 알고리즘은 모두 다수의 모형을 이용하지만 배깅은 모든 변수를 분석모형에 고려하고 있는 반면에, 랜덤 포레스트는 랜덤으로 변수를 추출하여 분석한다는 점에서 차이가 있음.
- R 에서 randomForest 패키지의 randomForest() 함수를 이용하면 랜덤 포레스트 모델을 학습할 수 있음.

참고자료(Reference)

1) https://ko.wikipedia.org/wiki/%EB%9E%9C%EB%8D%A4_%ED%8F%AC%EB%A0%88%EC%8A%A4%ED%8A%B8
2) https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/