

앙상블의 이해 & Bagging & Random Forest

5조

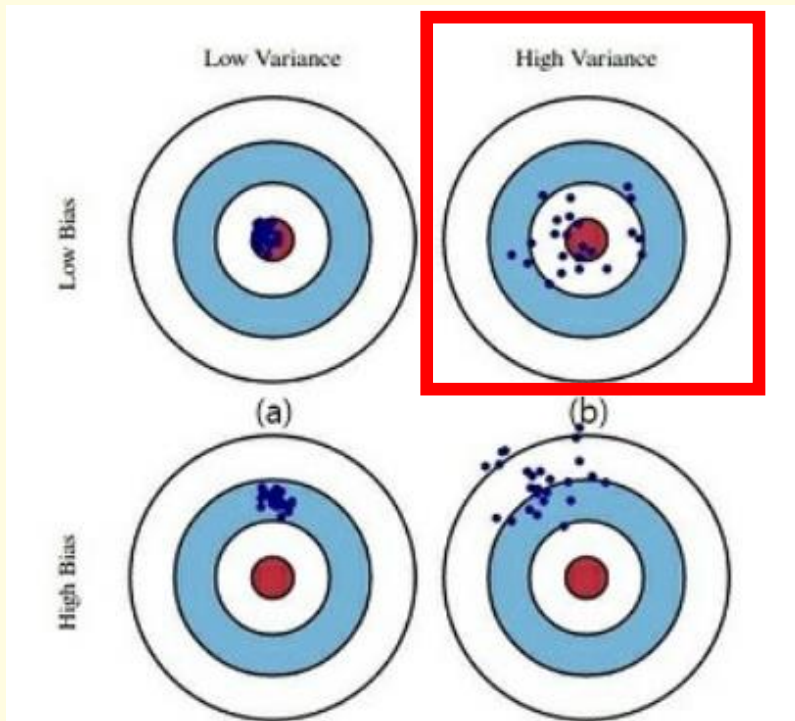
강한얼 김봉석 신은아

CONTENTS

- 01 의사결정나무의 단점
- 02 앙상블
- 03 Bootstrap
- 04 Bagging
- 05 Random Forest
- 06 실습
- 07 출처

01 의사결정나무의 단점

1. 계층적 구조로 인해 중간에 에러가 발생하면 다음 단계로 에러가 계속 전파
2. 학습 데이터의 미세한 변동에도 최종 결과가 크게 영향을 받음
3. 적은 개수의 노이즈에도 크게 영향
4. 나무의 최종 노드 개수를 늘리면 과적합 위험 발생 (Low Bias, Large Variance)
5. 가지치기 작업을 수행했음에도 과적합 되는 경향이 있어 모델의 일반화 성능이 좋지 못함



➡ **해결방안 : 랜덤 포레스트 모델 !!**

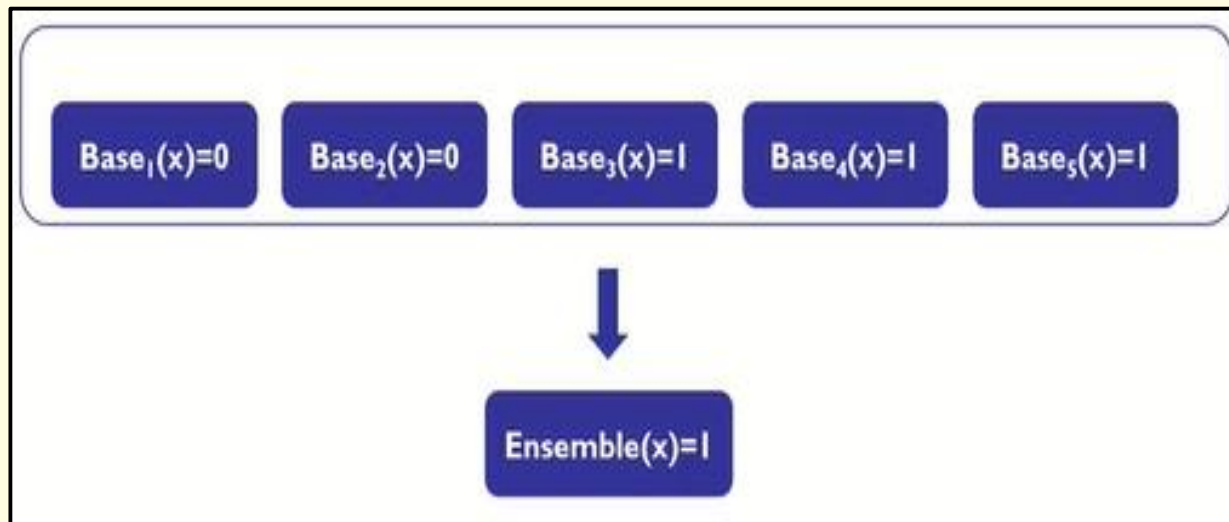
* bias는 참 값들과 추정 값들의 차이(or 평균간의 거리)를 의미하고,
variance는 추정 값들의 흩어진 정도를 의미

02 앙상블

- Ensemble(앙상블) : 프랑스어로 “통일” or “조화” 등을 의미함

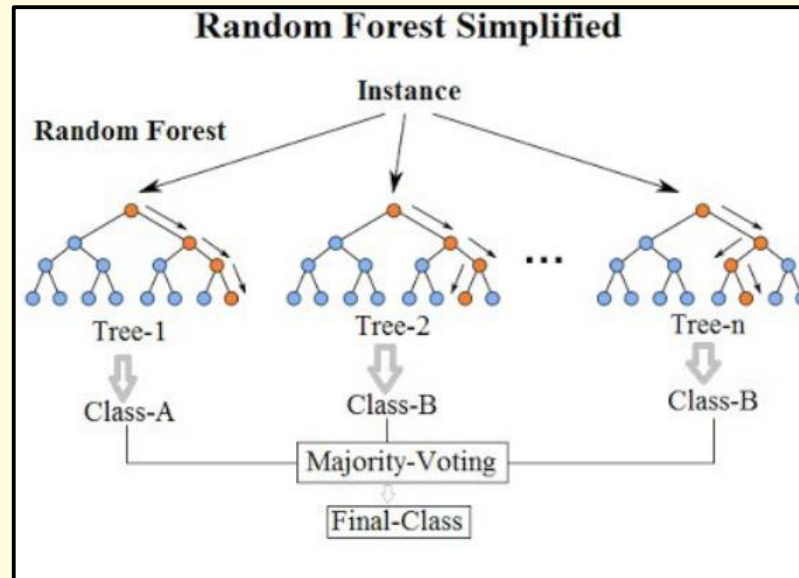
“두 사람 이상의 연주자에 의한 합주 또는 합창을 말하며, 같은 악기, 다른 악기에 관없이 듀엣에서 튜티라 불리는 총주까지 모두 앙상블에 속한다.”

→ 머신러닝에서는, 여러 Base모델들의 예측을 다수결 법칙 또는 평균을 이용해 통합하여 예측 정확성을 향상시키는 방법론을 말함



02 앙상블

- **앙상블 모형** : 여러 개의 분류모형에 의한 결과를 종합하여 분류의 정확도를 높이는 방법
 - 이는 적절한 표본추출법으로,
데이터에서 여러 개의 훈련용 데이터 집합을 만들고
 - > 각각의 데이터 집합에서 하나의 분류기를 만들어 앙상블을 하는 방법
 - 즉, 새로운 자료에 대해 분류기 예측값들의 가중 투표(weighted vote) 또는 다수결 투표(majority vote)를 통해 분류를 수행
- 데이터를 조절하는 가장 대표적인 방법에는 배깅(bagging)과 부스팅(boosting)이 있음
특히, 랜덤포레스트 방법은 배깅의 개념과 속성(변수)의 임의선택(random selection)을 결합한 앙상블 기법이다.



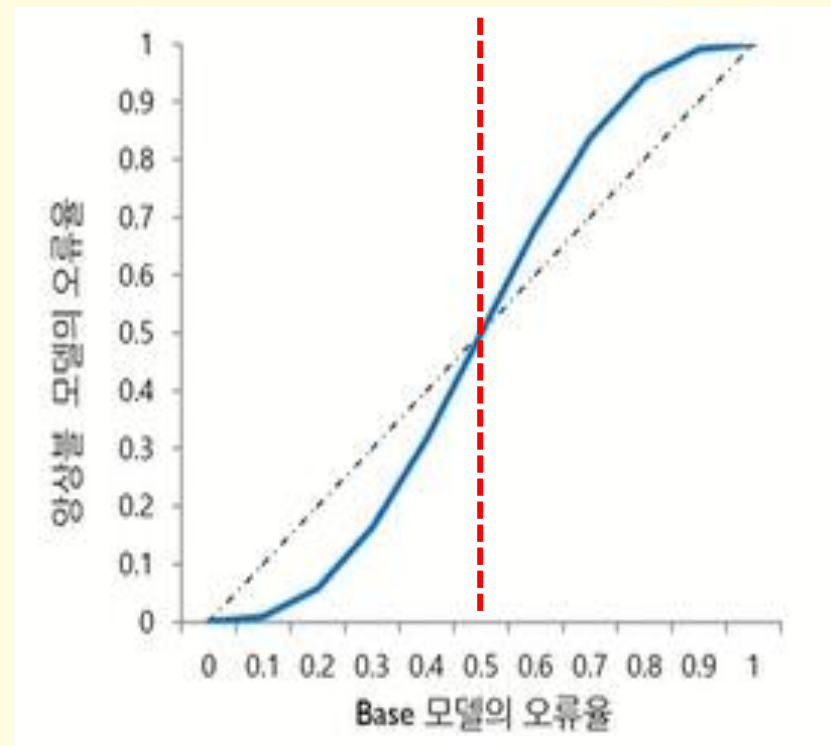
02 앙상블

다음 조건을 만족할 때 앙상블 모델은 Base 모델보다 우수한 성능을 보여준다 .

1. Base 모델들이 서로 독립적
2. Base 모델들이 무작위 예측을 수행하는 모델보다 성능이 좋은 경우
(ex 범주가 2개인 2 class 문제에 대해 최소한 0.5는 넘어야 한다.)

$$e_{ensemble} = \sum_{i=\lfloor N/2 \rfloor}^N \binom{N}{i} e^i (1-e)^{N-i}$$

e : Base 모델의 오류율
 N : Base 모델의 수



02 앙상블

- **앙상블의 장점**

- **평균을 취함으로써 편의를 제거해줌 :**

치우침이 있는 여러 모형의 평균을 취하면, 어느 쪽에도 치우치지 않는 결과(평균)를 얻게 된다.

- **분산을 감소시킴 :**

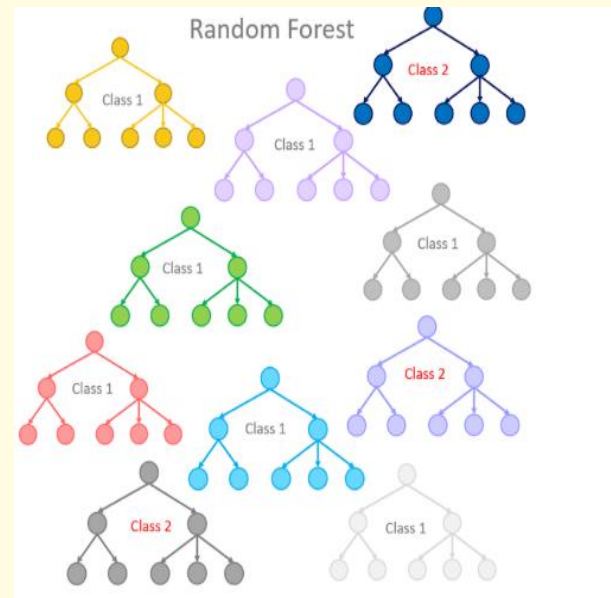
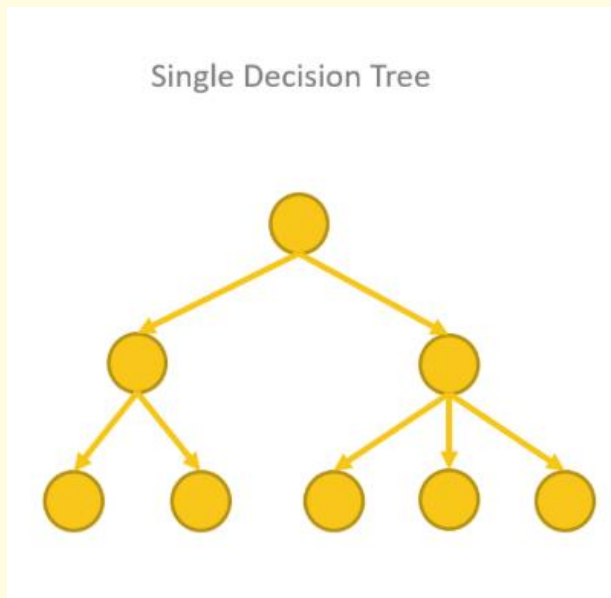
한 개 모형으로부터 단일 의견보다 여러 모형의 의견을 결합하면 변동이 작아진다.

- **과적합의 가능성을 줄여줌 :**

과적합이 없는 각 모형으로 부터 예측을 결합(평균, 가중 평균, 로지스틱 회귀)하면 과적합의 여지가 줄어든다.

02 앙상블

- 앙상블과 랜덤포레스트와의 관계



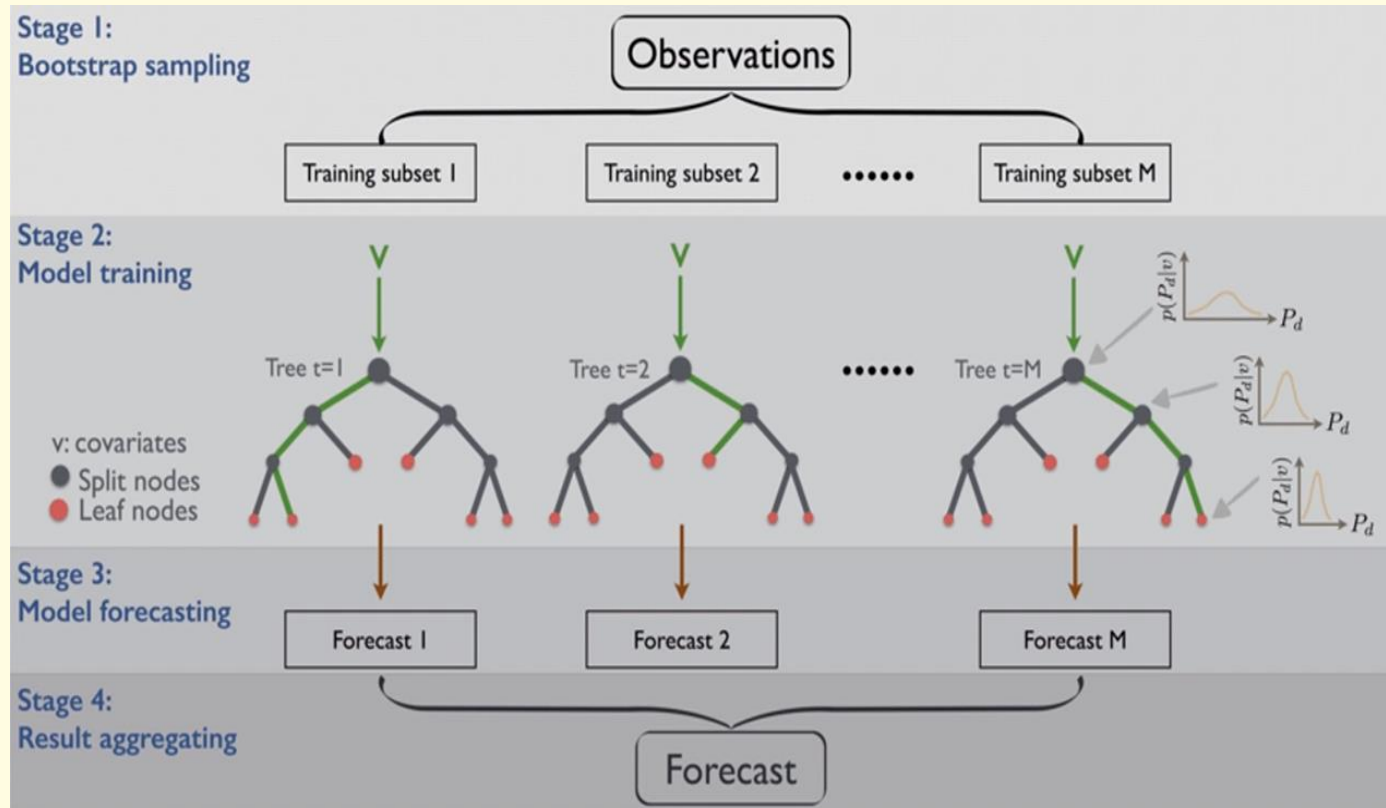
- 앙상블모델은 “어떤 Base 모델을 사용하느냐”에 따라 이름이 달라짐
→ 의사결정 나무를 base 모델로 사용하는 ensemble 기법을 Random Forest라고 한다.
- 의사결정 나무모델은 앙상블 모델의 base 모델로서 활용도가 높음
 - Low computational complexity : 데이터의 크기가 방대한 경우에도 모델을 빨리 구축
 - Nonparametric : 데이터 분포에 대한 전제가 필요하지 않음, 비모수적 모델

03 Bootstrap

- Bootstrap(부스트랩) : 주어진 자료에서 단순랜덤 복원추출 방법을 활용
→ 동일한 크기의 표본을 여러 개 생성하는 샘플링 방법
- 개별 데이터셋을 '부스트랩셋'이라고 부름

Original Dataset	Bootstrap 1	Bootstrap 2		Bootstrap B																																																																																
<table><tr><td>x^1</td><td>y^1</td></tr><tr><td>x^2</td><td>y^2</td></tr><tr><td>x^3</td><td>y^3</td></tr><tr><td>x^4</td><td>y^4</td></tr><tr><td>x^5</td><td>y^5</td></tr><tr><td>x^6</td><td>y^6</td></tr><tr><td>x^7</td><td>y^7</td></tr><tr><td>x^8</td><td>y^8</td></tr><tr><td>x^9</td><td>y^9</td></tr><tr><td>x^{10}</td><td>y^{10}</td></tr></table>	x^1	y^1	x^2	y^2	x^3	y^3	x^4	y^4	x^5	y^5	x^6	y^6	x^7	y^7	x^8	y^8	x^9	y^9	x^{10}	y^{10}	<table><tr><td>x^3</td><td>y^3</td></tr><tr><td>x^6</td><td>y^6</td></tr><tr><td>x^2</td><td>y^2</td></tr><tr><td>x^{10}</td><td>y^{10}</td></tr><tr><td>x^8</td><td>y^8</td></tr><tr><td>x^7</td><td>y^7</td></tr><tr><td>x^7</td><td>y^7</td></tr><tr><td>x^3</td><td>y^3</td></tr><tr><td>x^2</td><td>y^2</td></tr><tr><td>x^7</td><td>y^7</td></tr></table>	x^3	y^3	x^6	y^6	x^2	y^2	x^{10}	y^{10}	x^8	y^8	x^7	y^7	x^7	y^7	x^3	y^3	x^2	y^2	x^7	y^7	<table><tr><td>x^7</td><td>y^7</td></tr><tr><td>x^1</td><td>y^1</td></tr><tr><td>x^{10}</td><td>y^{10}</td></tr><tr><td>x^1</td><td>y^1</td></tr><tr><td>x^8</td><td>y^8</td></tr><tr><td>x^6</td><td>y^6</td></tr><tr><td>x^2</td><td>y^2</td></tr><tr><td>x^6</td><td>y^6</td></tr><tr><td>x^4</td><td>y^4</td></tr><tr><td>x^9</td><td>y^9</td></tr></table>	x^7	y^7	x^1	y^1	x^{10}	y^{10}	x^1	y^1	x^8	y^8	x^6	y^6	x^2	y^2	x^6	y^6	x^4	y^4	x^9	y^9	...	<table><tr><td>x^9</td><td>y^9</td></tr><tr><td>x^5</td><td>y^5</td></tr><tr><td>x^2</td><td>y^2</td></tr><tr><td>x^4</td><td>y^4</td></tr><tr><td>x^7</td><td>y^7</td></tr><tr><td>x^2</td><td>y^2</td></tr><tr><td>x^5</td><td>y^5</td></tr><tr><td>x^{10}</td><td>y^{10}</td></tr><tr><td>x^8</td><td>y^8</td></tr><tr><td>x^2</td><td>y^2</td></tr></table>	x^9	y^9	x^5	y^5	x^2	y^2	x^4	y^4	x^7	y^7	x^2	y^2	x^5	y^5	x^{10}	y^{10}	x^8	y^8	x^2	y^2
x^1	y^1																																																																																			
x^2	y^2																																																																																			
x^3	y^3																																																																																			
x^4	y^4																																																																																			
x^5	y^5																																																																																			
x^6	y^6																																																																																			
x^7	y^7																																																																																			
x^8	y^8																																																																																			
x^9	y^9																																																																																			
x^{10}	y^{10}																																																																																			
x^3	y^3																																																																																			
x^6	y^6																																																																																			
x^2	y^2																																																																																			
x^{10}	y^{10}																																																																																			
x^8	y^8																																																																																			
x^7	y^7																																																																																			
x^7	y^7																																																																																			
x^3	y^3																																																																																			
x^2	y^2																																																																																			
x^7	y^7																																																																																			
x^7	y^7																																																																																			
x^1	y^1																																																																																			
x^{10}	y^{10}																																																																																			
x^1	y^1																																																																																			
x^8	y^8																																																																																			
x^6	y^6																																																																																			
x^2	y^2																																																																																			
x^6	y^6																																																																																			
x^4	y^4																																																																																			
x^9	y^9																																																																																			
x^9	y^9																																																																																			
x^5	y^5																																																																																			
x^2	y^2																																																																																			
x^4	y^4																																																																																			
x^7	y^7																																																																																			
x^2	y^2																																																																																			
x^5	y^5																																																																																			
x^{10}	y^{10}																																																																																			
x^8	y^8																																																																																			
x^2	y^2																																																																																			

04 Bagging



* aggregate : 종합하다

• Bagging(Bootstrap Aggregating)의 과정

1. 여러 개의 붓스트랩 자료를 생성
2. 각 붓스트랩 자료에 예측모형을 만듦
3. 예측된 결과를 결합
4. 최종 예측모형을 만듦

* 최적의 의사결정나무를 구축할 때 가장 어려운 부분 : 가지치기(pruning). But, 배경에서는 가지치기를 하지 않고 최대로 성장한 의사결정나무들을 활용

04 Bagging

- **Bagging의 효과**

여러 예측 모형의 결과를 결합하므로 배깅은 **Variance**를 낮춰주는데 도움을 줌

→ Large Variance를 가진 모델에 유용하게 잘 사용됨

* Variance : training set(모델 학습에서 사용된 data set)에 따라 결과가 달라지는 정도.
모형의 flexibility와 관련이 있음

→ Large Variance : training set에 사소한 변화가 있을 때, 예측 결과가 크게 달라진다는 의미

04 Bagging

- 의사결정나무의 특징 : **Large Variance**, Low bias

→ Variance를 낮춰줄 필요가 있겠네?!

→ 배깅은 의사결정나무와 자주 같이 사용됨

- Bagging을 의사결정나무에 적용해보자!

1. B개의 붓스트랩 자료를 생성

2. 각 붓스트랩 자료에 대해 B개의 의사결정나무 생성

(여기서 각각의 의사결정 나무는 Large Variance, Low bias의 특징을 가짐)

3. 예측된 결과를 결합

회귀트리 : 예측값의 평균을 냄

분류트리 : (1) Majority voting

(2) Weighted voting(weight = training accuracy of individual models)

(3) Weighted voting(weight=predicted probability for each class)

4. 최종 예측모형을 만듦

다음 페이지에서 자세히
알아봅시다 ^_^

04 Bagging

* voting : 여러 개의 모형으로부터 산출된 결과에서 다수결에 의해 최종 결과를 선정하는 과정

• 분류트리에서 결과를 결합하는 방법(1) Majority voting

- Majority voting

$$Ensemble(\hat{y}) = \underset{i}{\operatorname{argmax}} \left(\sum_{j=1}^n I(\hat{y}_j = i), i \in \{0, 1\} \right)$$

$$\sum_{j=1}^n I(\hat{y}_j = 0) = 4$$

$$\sum_{j=1}^n I(\hat{y}_j = 1) = 6$$

지시함수는 조건을 만족하면 1, 아니면 0
→ 왼쪽의 식은 각각 $\hat{y}_j = 0, 1$ 인 것의 개수이고, \hat{y}_j 는 Predicted class label임
→ 왼쪽 그림의 Predicted class label을 보면 0이 4개, 1이 6개임

Training Accuracy	Ensemble population	P(y=1) for a test instance	Predicted class label
0.80	Model 1	0.90	1
0.75	Model 2	0.92	1
0.88	Model 3	0.87	1
0.91	Model 4	0.34	0
0.77	Model 5	0.41	0
0.65	Model 6	0.84	1
0.95	Model 7	0.14	0
0.82	Model 8	0.32	0
0.78	Model 9	0.98	1
0.83	Model 10	0.57	1

$P(y=1) > 0.5 \rightarrow$ Predicted class label=1
 $P(y=1) < 0.5 \rightarrow$ Predicted class label=0

Ensemble(\hat{y})은 식에서 괄호안을 최대로 하는 i 를 찾아야함
→ $i=0$ 일때 괄호안의 식은 4 < $i=1$ 일때 괄호안의 식은 6
→ Ensemble(\hat{y})=1

$$Ensemble(\hat{y}) = 1$$

04 Bagging

• 분류트리에서 결과를 결합하는 방법(2) Weighted voting(weight = training accuracy of individual models)

- Weighted voting (weight = training accuracy of individual models)

$$Ensemble(\hat{y}) = \underset{i}{\operatorname{argmax}} \left(\frac{\sum_{j=1}^n (TrainAcc_j) \cdot I(\hat{y}_j = i)}{\sum_{j=1}^n (TrainAcc_j)}, i \in \{0, 1\} \right)$$

이전의 방법처럼 Predicted class label이 0인지 1인지만 보는 것이 아니라, Training Accuracy로 가중평균을 내는 방법

Training Accuracy	Ensemble population	P(y=1) for a test instance	Predicted class label
0.80	Model 1	0.90	1
0.75	Model 2	0.92	1
0.88	Model 3	0.87	1
0.91	Model 4	0.34	0
0.77	Model 5	0.41	0
0.65	Model 6	0.84	1
0.95	Model 7	0.14	0
0.82	Model 8	0.32	0
0.78	Model 9	0.98	1
0.83	Model 10	0.57	1

$$\frac{\sum_{j=1}^n (TrainAcc_j) \cdot I(\hat{y}_j = 0)}{\sum_{j=1}^n (TrainAcc_j)} = 0.424$$

$$\frac{\sum_{j=1}^n (TrainAcc_j) \cdot I(\hat{y}_j = 1)}{\sum_{j=1}^n (TrainAcc_j)} = 0.576$$

$$Ensemble(\hat{y}) = 1$$

위의 식들은 세션 때 같이 계산해봅시다^0^

04 Bagging

• 분류트리에서 결과를 결합하는 방법(3) Weighted voting(weight=predicted probability for each class)

- Weighted voting (weight = predicted probability for each class)

$$Ensemble(\hat{y}) = \underset{i}{\operatorname{argmax}} \left(\frac{1}{n} \sum_{j=1}^n P(y = i), i \in \{0, 1\} \right)$$

$$\frac{1}{10} \sum_{j=1}^{10} P(y = 0) = 0.371$$

$$\frac{1}{10} \sum_{j=1}^{10} P(y = 1) = 0.629$$

주황색 박스는 모두 $P(y=1)$ 의 값
 → 왼쪽의 식을 계산해보면, 각각
 $\frac{1}{10} \{(1-0.90)+(1-0.92)+\dots+(1-0.57)\}=0.371$,
 $\frac{1}{10} (0.90+0.92+\dots+0.57)=0.629$ 임

Training Accuracy	Ensemble population	$P(y=1)$ for a test instance	Predicted class label
0.80	Model 1	0.90	1
0.75	Model 2	0.92	1
0.88	Model 3	0.87	1
0.91	Model 4	0.34	0
0.77	Model 5	0.41	0
0.65	Model 6	0.84	1
0.95	Model 7	0.14	0
0.82	Model 8	0.32	0
0.78	Model 9	0.98	1
0.83	Model 10	0.57	1




Ensemble(\hat{y})은 식에서 괄호안을 최대로 하는 i 를 찾아야함
 → $i=0$ 일때 괄호안의 식은 $0.371 < i=1$ 일때 괄호안의 식은 0.629
 → Ensemble(\hat{y})=1

$$Ensemble(\hat{y}) = 1$$

05 Random Forest

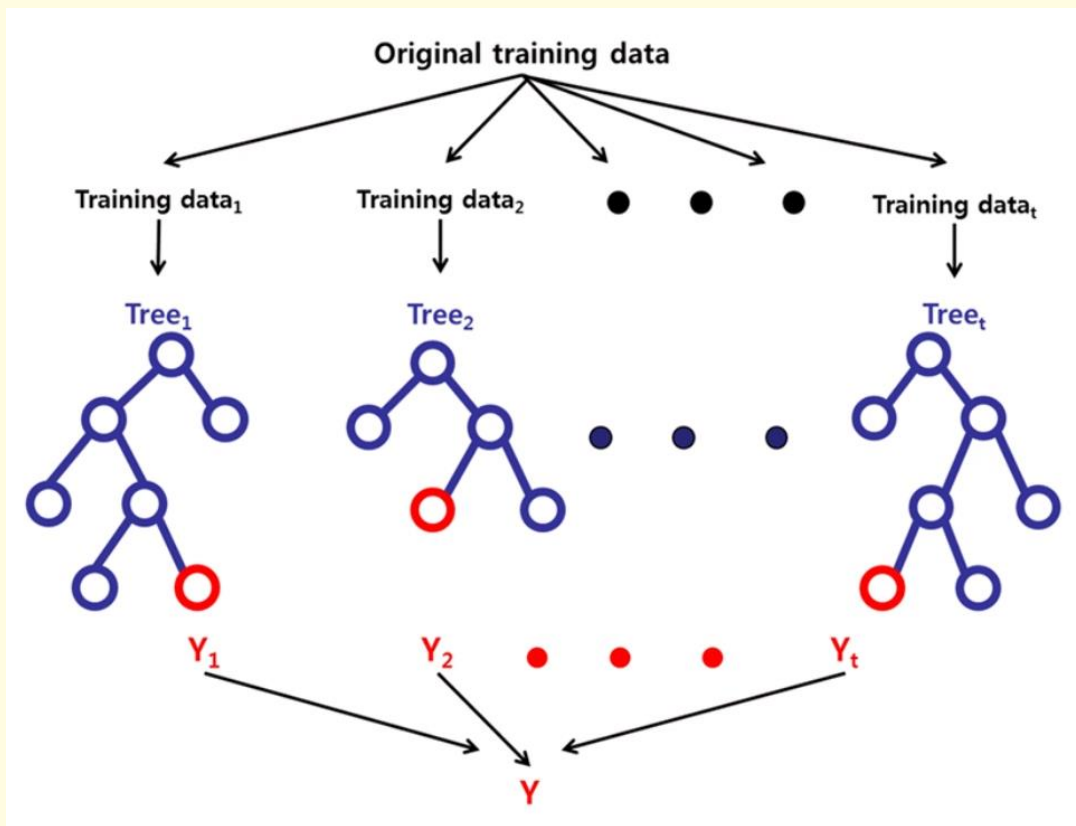
Random Forest = Bagging + Random subspace

- **Bagging** : 여러 개의 training data 생성하여 각 데이터마다 개별 의사결정나무모델 구축
→ **Diversity** 확보
- **Random subspace** : 의사결정나무모델 구축 시 변수 무작위로 선택
→ **Random** 확보

 **Random Forest의 핵심 아이디어 : Diversity, Random 확보하는 것!**

05 Random Forest

- Random Forest(랜덤 포레스트) : 다수의 의사결정나무모델에 의한 예측을 종합하는 앙상블 방법



- Random Forest의 과정

1. Bootstrap 기법을 활용하여 다수의 training data 생성
2. 생성된 training data로 의사결정나무 모델 구축 (무작위 변수 사용)
3. 예측 종합

- Random Forest의 활용

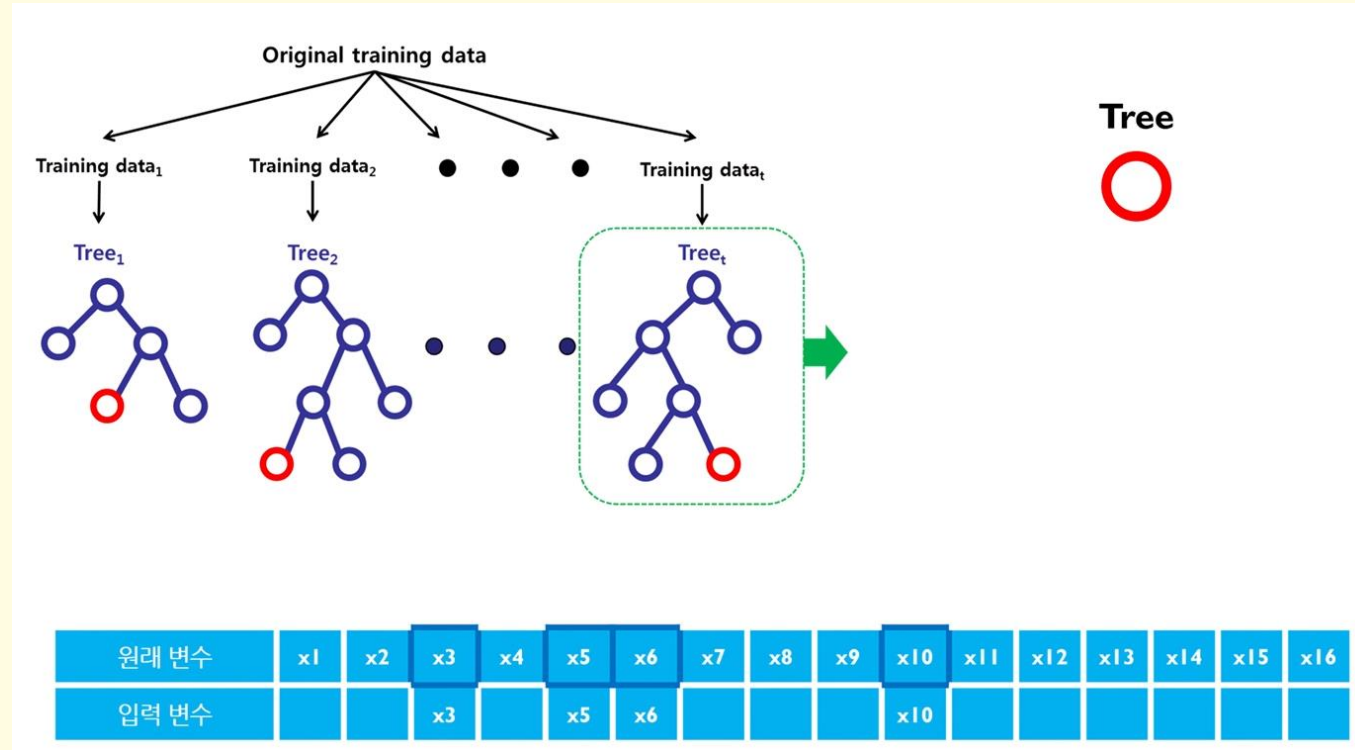
관측치 수에 비해 변수의 수가 많은 고차원 데이터에서 중요 변수 선택 기법으로 널리 활용

ex) 건강의 위험도를 예측하기 위해서는 성별, 키, 운동량, 흡연유무, 기초 대사량 등 수많은 요소를 고려해야 함. 이러한 수많은 요소를 기반으로 건강의 위험도를 예측하는 하나의 결정 트리를 만든다면 트리의 가지가 많아 질 것 → 오버피팅

하지만, 랜덤 포레스트를 이용하여 수많은 요소 중 랜덤으로 5개의 요소만 선택해서 하나의 결정 트리를 만들고, 또 5개의 요소만 선택해서 또 다른 결정 트리를 만들고, ... 이렇게 계속 반복하여 만들어진 결정트리들의 예측값들을 종합하면 효율적임

05 Random Forest

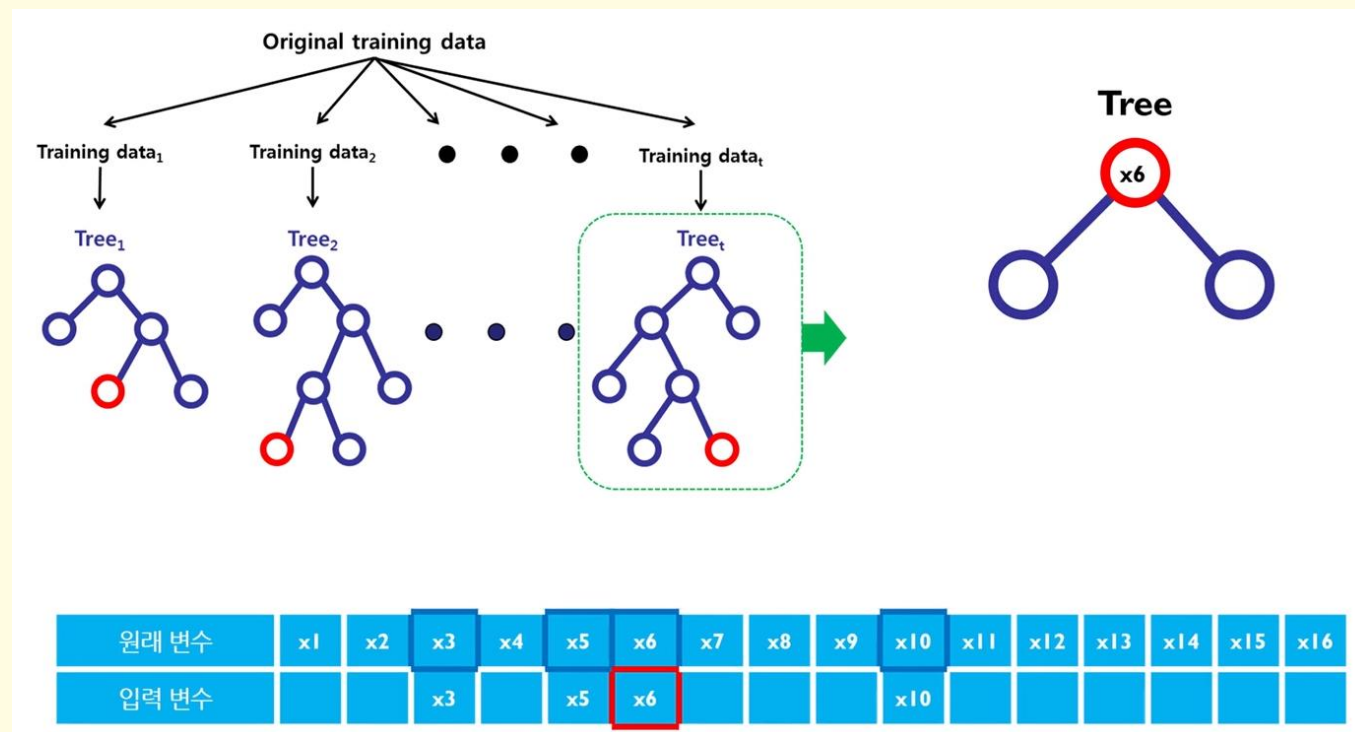
- Random subspace



- 원래 변수들 중에서 모델 구축에 쓰일 입력 변수를 무작위로 선택
(의사결정나무의 분기점을 탐색할 때, 원래 변수의 수보다 적은 수의 변수를 임의로 선택 → 해당 변수들만을 고려 대상으로 함)

05 Random Forest

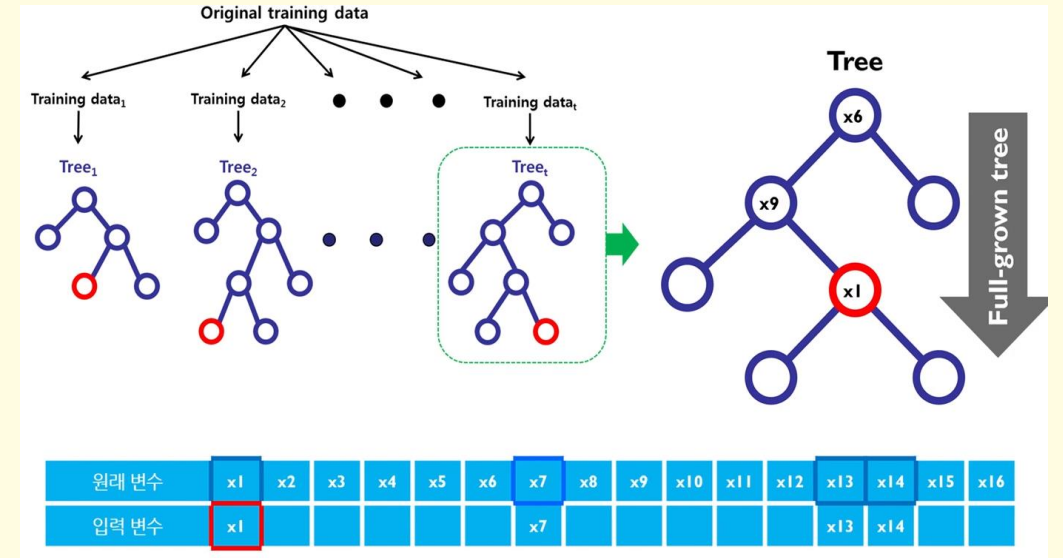
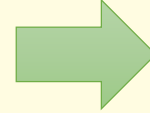
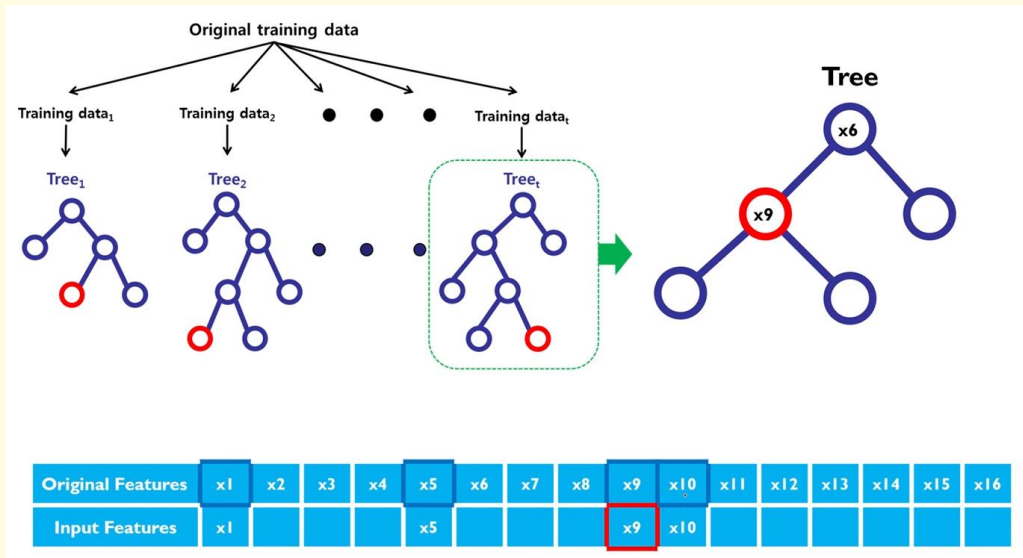
- Random subspace



2. 선택된 입력 변수 중 분할된 변수를 선택

05 Random Forest

- Random subspace



3. 이러한 과정을 full grown tree가 될 때까지 반복

05 Random Forest

- **Random subspace가 필요한 이유?**

트리들 간의 상관성을 없애기 위해!

(소수의 특징들이 결과에 대해 강한 예측 성능을 지닌다면,

훈련 과정 중 여러 트리 노드에서 이러한 특징들이 중복되어 선택됨 → 결과적으로 트리들이 상관성을 가짐.

즉, 소수의 특징들이 데이터를 분류함에 있어서 중요한 기준이 된다면 → 한 트리 뿐만 아니라 여러 트리에서 중요한 특징이 될 것이고, 상관성을 가질 것임)

05 Random Forest

- Random Forest의 중요변수선택

- 랜덤 포레스트(순수한 비모수적 모델)는 선형 회귀모델 or 로지스틱 회귀모델과는 달리 개별 변수가 통계적으로 얼마나 유의한지에 대한 정보 제공X(← 의사결정나무가 알려진 확률분포를 가정하지 않기 때문)

- * 확률분포를 가정 → 통계적인 추론 가능(통계적으로 중요하다, 안 중요하다를 해석 가능)

- 대신 랜덤 포레스트는 다음과 같은 간접적인 방식으로 변수의 중요도를 결정

- 1. 원래 데이터 집합에 대해서 OOB Error를 구함

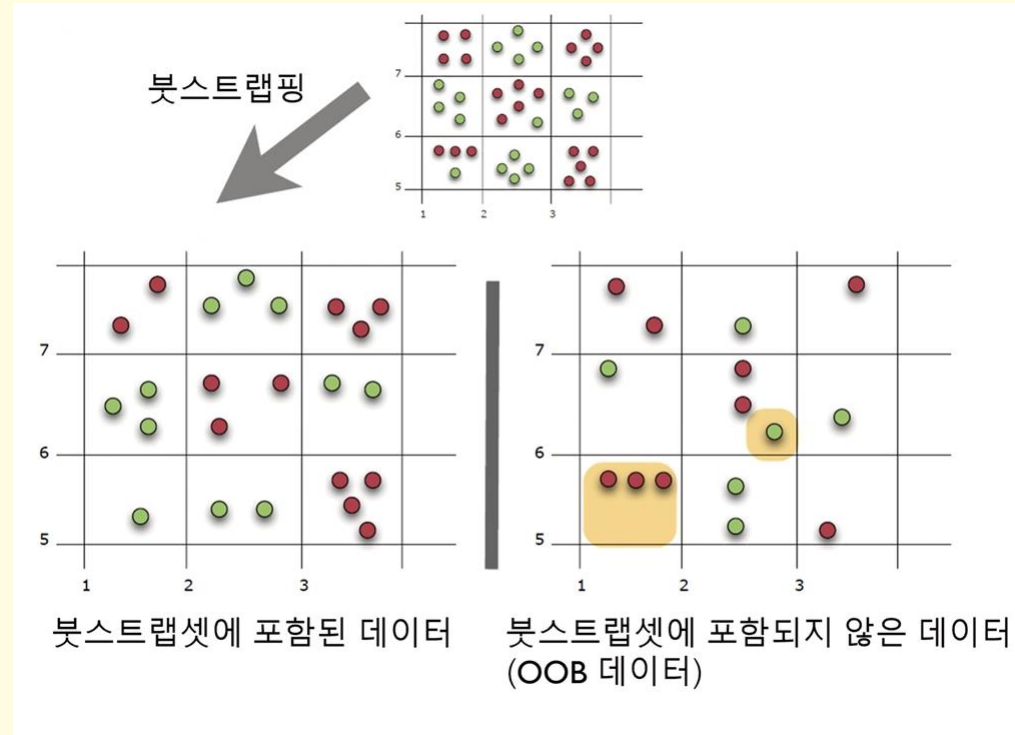
- 2. 특정 변수의 값을 임의로 뒤섞은 데이터 집합에 대해 OOB Error를 구함

- 3. 개별 변수의 중요도는 2단계와 1단계의 OOB Error차이의 평균과 분산을 고려하여 결정

처음 보는 용어가 나왔죠?!
다음 페이지에서 자세히 알아보시다 ^_ ^

05 Random Forest

- Random Forest의 중요변수선택



* OOB(Out of Bag) 데이터 : 붓스트랩에 포함되지 않은 데이터

✖✖ 잠깐 다시 Bootstrap으로!

- 이론적으로 한 개체가 하나의 붓스트랩에 한번도 선택되지 않을 확률 :

$$p = \left(1 - \frac{1}{N}\right)^N \rightarrow \lim_{N \rightarrow \infty} \left(1 - \frac{1}{N}\right)^N = e^{-1} = 0.368$$

→ 이는 붓스트랩을 통해 100개의 샘플을 추출하더라도 샘플에 한번도 선택되지 않는 원데이터(약 36.8%)가 발생할 수 있음을 의미

↓
요게 바로 OOB 데이터!

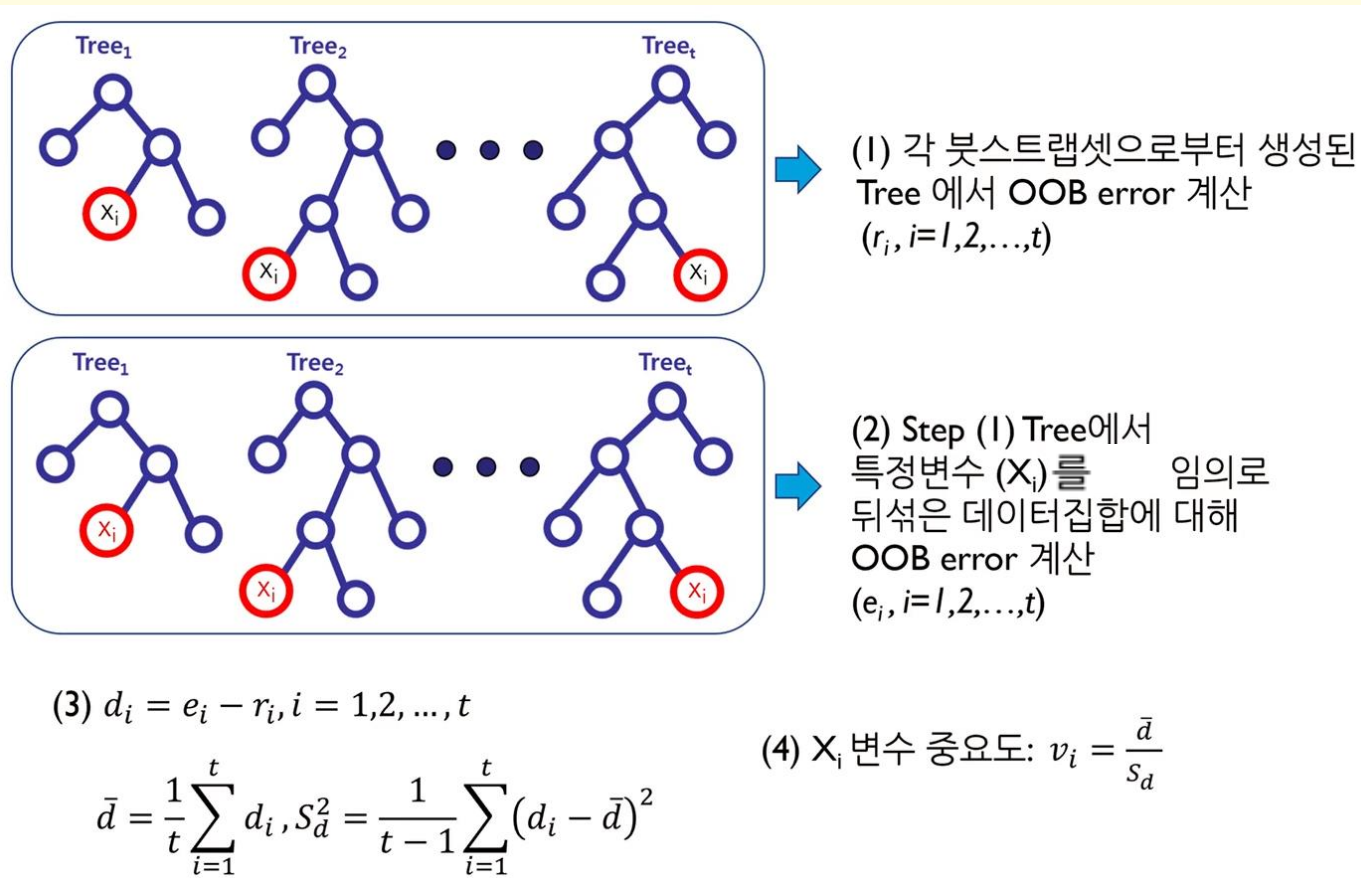
- 배깅을 사용할 경우 OOB 데이터들을 검증 집합으로 사용

05 Random Forest

• Random Forest의 중요변수선택

* X_i (i번째 변수)의 중요도를 보는 과정

* OOB error : OOB 데이터를 사용했을 때의 error



- d_i 가 크다 : 해당 변수가 다른 변수로 대체됐을 때 성능이 안좋아진다는 의미
→ 해당 변수가 중요한 변수라는 소리
→ \bar{d} 는 X_i 변수 중요도에서 분자에 위치
- X_i 변수 중요도에서 분모에 해당하는 S_d 는 패널티를 주는 역할을 함.
→ d 의 표준편차가 크면 d 의 평균이 크더라도, 표준편차가 크면서 큰 것이기 때문에 약간의 패널티를 주겠다.
- v_i 가 커지면 → 변수의 중요도 커짐

05 Random Forest

- Random Forest의 hyperparameter

* rough한 가이드라인일뿐, 정해져 있는 것은 아님

1. Decision tree의 수

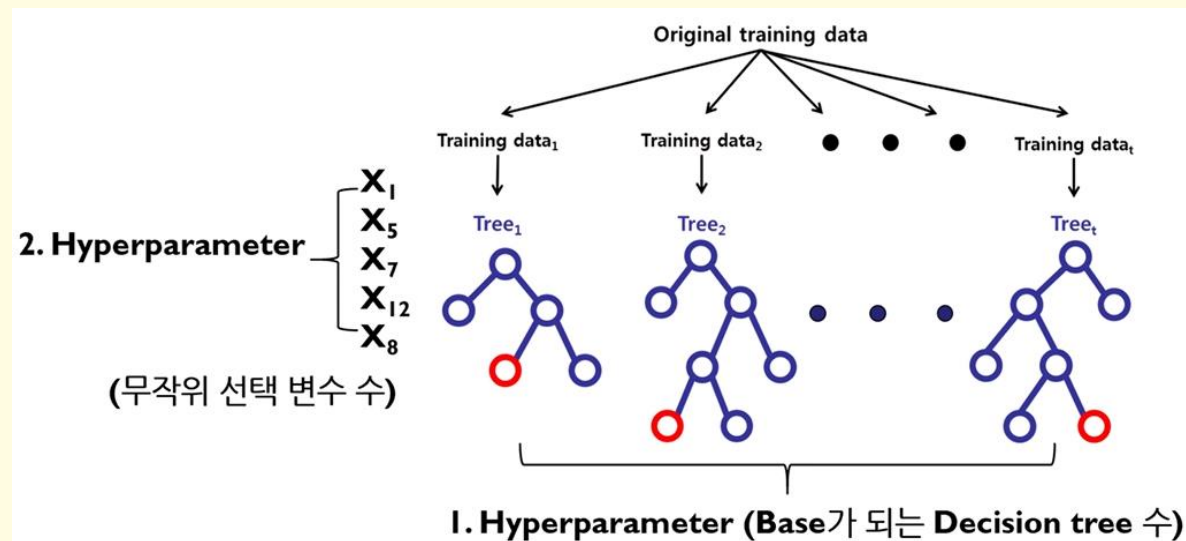
Strong law of large numbers를 만족시키기 위해 2,000개 이상의 decision tree 필요

2. Decision tree에서 노드 분할 시 무작위로 선택되는 변수의 수

일반적으로 변수의 수에 따라 다음과 같이 추천됨

Classification 모델에서 : $\sqrt{\text{변수의 수}}$

Regression 모델에서 : 변수의 수 / 3



05 Random Forest

- **Random Forest의 장&단점**

장점

- 관측치 수에 비해 변수의 수가 많은 고차원 데이터에서 중요 변수 선택 기법으로 널리 활용
- 일반적으로 하나의 의사결정나무모델보다 높은 예측 정확성을 보여줌
- 알고리즘이 굉장히 간단함
(의사결정나무를 어떻게 만드는지만 알고 있다면, 이들의 결과를 종합하기만 하면 되기 때문에 굉장히 쉽게 이해하고 구현할 수 있음)
- 예측의 변동성이 줄어들며, 과적합 방지할 수 있음

단점

- 메모리 사용량이 굉장히 많음. 의사결정나무를 만드는 것으로만도 메모리를 많이 사용하는데, 이들을 여러 개 만들어 종합해야 하므로 memory consumption이 많음
- 데이터의 수가 많아지면 의사 결정나무에 비해 속도가 크게 떨어짐

06 실습

실습 순서

1. `randomForest()` 실행
2. 회귀트리 **vs** 랜덤포레스트
3. 배깅 **vs** 랜덤포레스트
4. 변수 중요도
5. Hyperparameter Tuning
6. `range()`

□ Boston Data Set in library(MASS)

```
2
3 #0. 데이터 불러오기
4 library(MASS)
5 library(caret)
6
7 set.seed(1234)
8 train_idx <- createDataPartition(Boston$medv, p = 0.7, list = F)
9
10 Boston_train <- Boston[train_idx,]
11 Boston_test <- Boston[-train_idx,]
12
13
```

Target	Description
medv	1978년 보스턴 주택 가격의 중앙값(단위 1,000\$)
Feature	Description
crim	Town별 1인당 범죄율
zn	25,000평방피트를 초과하는 거주지역의 비율
Indus	비소매상업지역이 점유하고 있는 토지의 비율
⋮	⋮
lstat	하위계층의 비율

06 실습 - 1. randomForest() 실행

□ 문법은 회귀모델 lm(종속변수~독립변수, data)와 유사합니다.

```
실습코드.R* x  Untitled1* x
Source on Save
14 #1. randomForest 실행
15 library(randomForest)
16 set.seed(1234)
17 Boston_rf <- randomForest(medv ~ ., data=Boston_train)
18 Boston_rf
19

Console  Terminal x  Jobs x
~/MyRHome/ ↗
> Boston_rf

Call:
randomForest(formula = medv ~ ., data = Boston_train)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 4

Mean of squared residuals: 12.18859
% Var explained: 86.23
>
```

“ ntree 와 mtry의 효과를 다른 모형과
비교해 보면서 알아보겠습니다! ”



- 1. **ntree** : 개별 트리의 수
(default = 500)
- 2. **mtry** : 노드 분할마다 고려되는 변수의 수
(default = (변수의 수) / 3)
- 3. **MSE** : Out of Bag Data를 사용하여 계산
- 3. **R^2** : $1 - \text{MSE} / S_{yy}$

06 실습 - 2. 회귀트리 vs 랜덤포레스트 : ntree=1 vs ntree=500

1

```
23 #2. 회귀트리 vs 랜덤포레스트
24 library(rpart)
25 set.seed(1234)
26 Boston_reg.tr <- rpart(medv ~ ., data=Boston_train)
27 Boston_reg.tr_pred <- predict(Boston_reg.tr, Boston_test)
28 Boston_rf_pred <- predict(Boston_rf, Boston_test)
29
30 MSE <- function(actual, predicted){
31   mean((actual - predicted)^2)
32 }
33
34 MSE(Boston_test$medv, Boston_reg.tr_pred)
35 MSE(Boston_test$medv, Boston_rf_pred)
```

2

```
Console Terminal x Jobs x
~/MyRHome/ ↗
> MSE(Boston_test$medv, Boston_reg.tr_pred)
[1] 20.33966
> MSE(Boston_test$medv, Boston_rf_pred)
[1] 6.324475
>
```

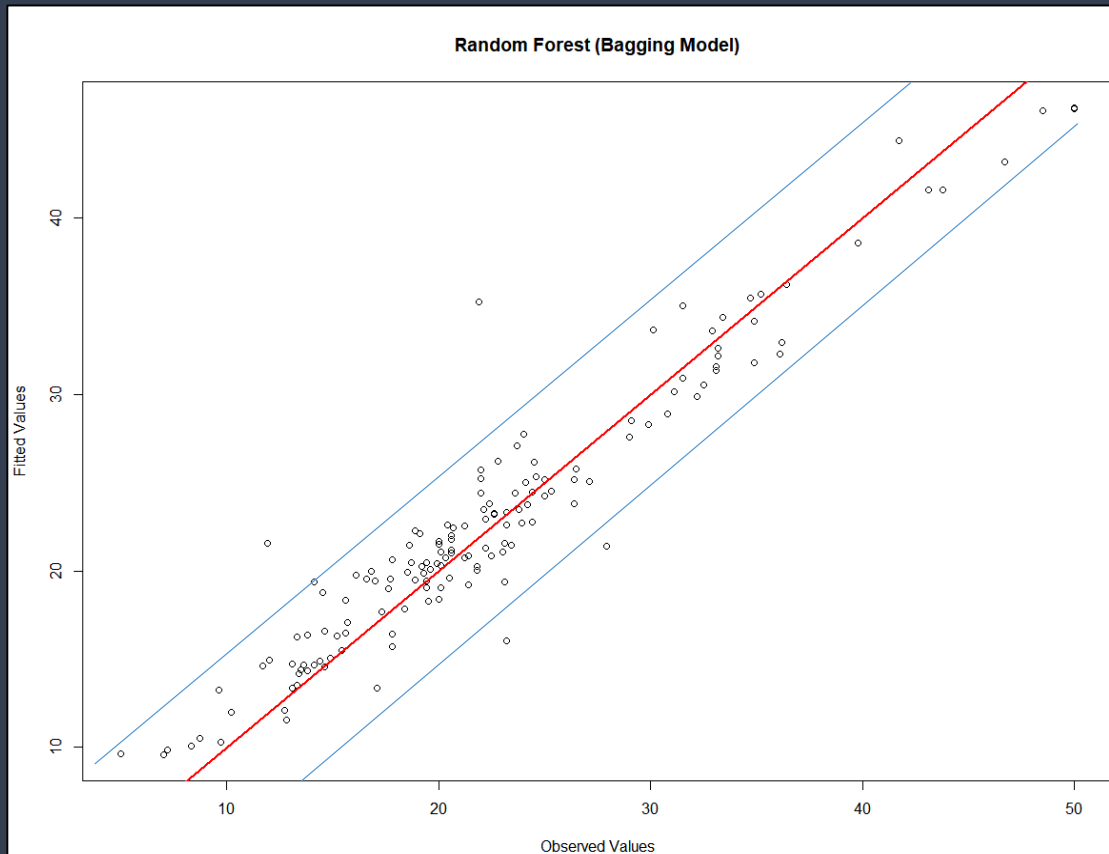
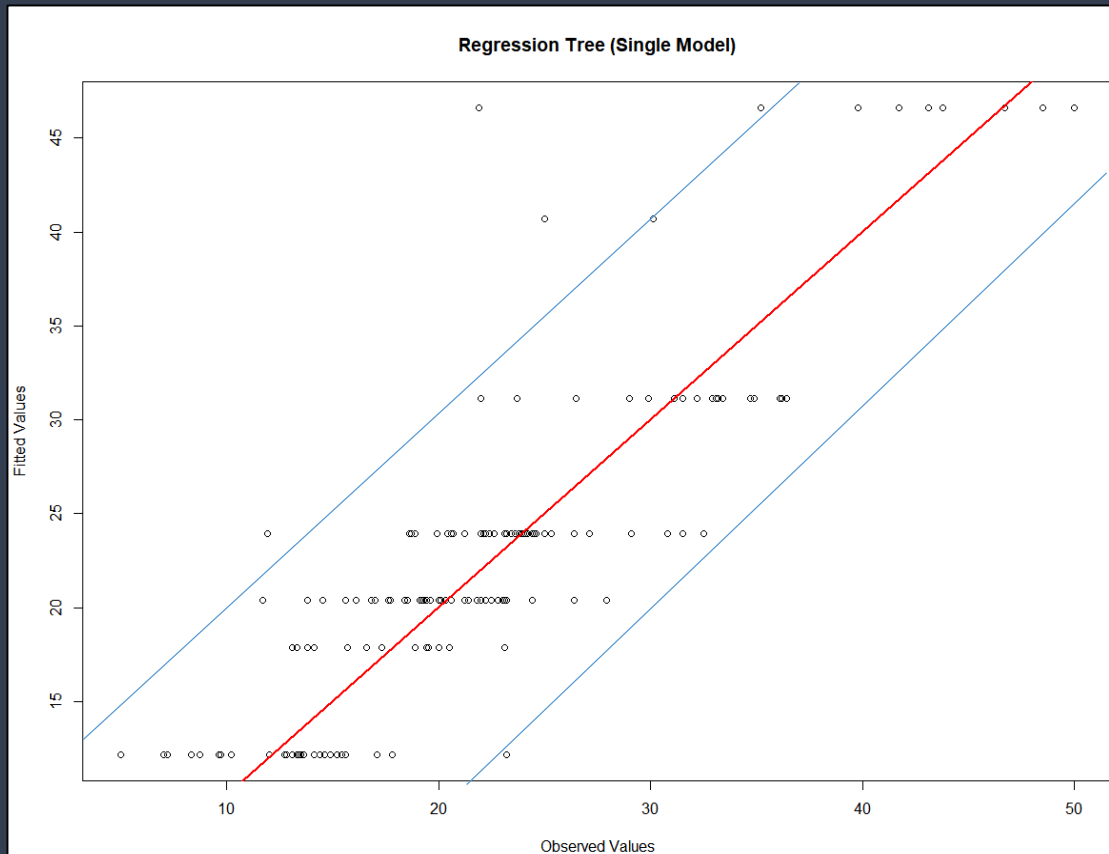
랜덤포레스트의 MSE가 단일 트리의 MSE보다 작음을 확인할 수 있습니다.

3

```
41 #회귀트리 모형
42 plot(Boston_test$medv, Boston_reg.tr_pred,
43       xlab = 'Observed Values', ylab = 'Fitted Values', main = 'Regression Tree (Single Model)')
44 abline(0, 1, col = 'red', lwd = 2)
45
46 #랜덤포레스트 모형
47 plot(Boston_test$medv, Boston_rf_pred,
48       xlab = 'Observed Values', ylab = 'Fitted Values', main = 'Random Forest (Bagging Model)')
49 abline(0, 1, col = 'red', lwd = 2)
```

Plot으로 확인해 볼게요
결과는 다음 슬라이드!

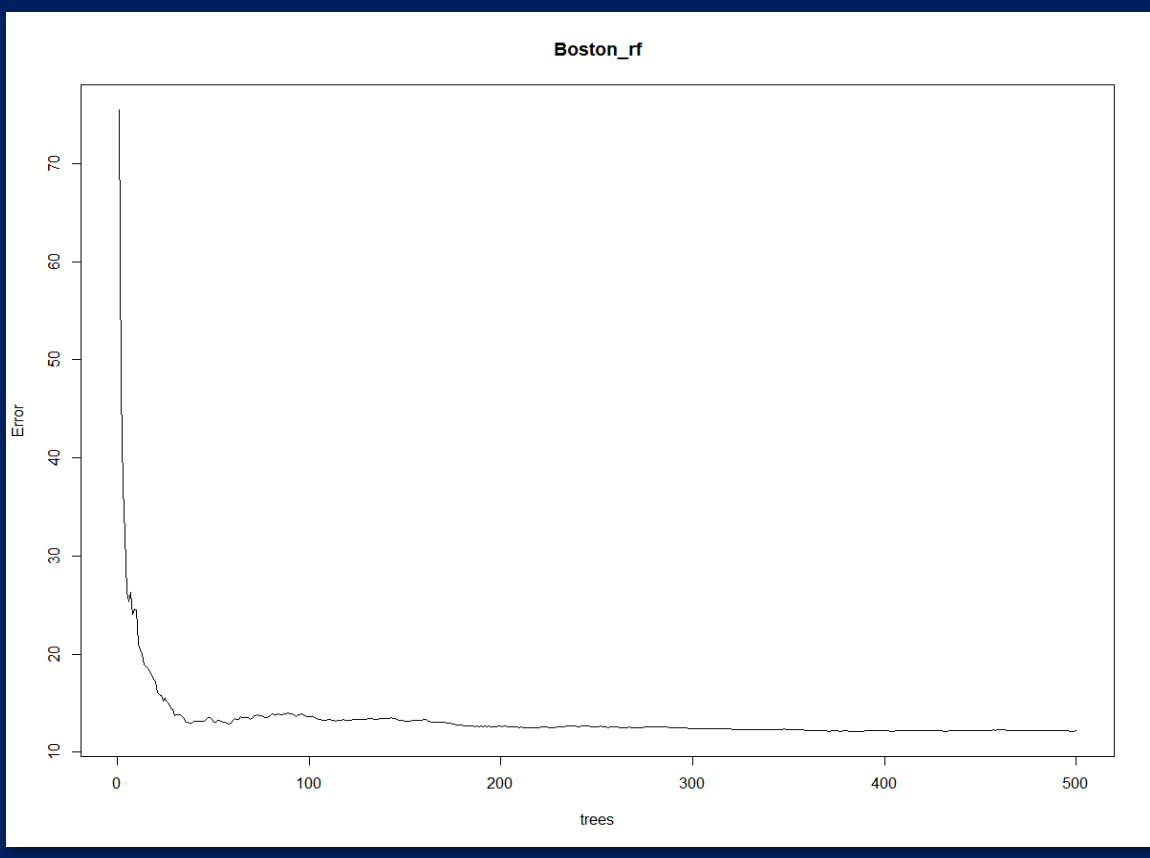
06 실습 - 2. 회귀트리 vs 랜덤포레스트 : ntree=1 vs ntree=500



두 결과 모두 편향되지 않은 것으로 보이지만 랜덤포레스트로 예측한 결과의 분산이 훨씬 작은 것을 확인할 수 있죠.

06 실습 - 2. 회귀트리 vs 랜덤포레스트 : ntree=1 vs ntree=500

```
실습코드.R* x Untitled1* x  
← → Source on Save  
21 plot(Boston_rf)  
22  
23
```



$$\hat{y}_i \sim \mathcal{D}(\mu, \sigma^2)$$
$$\bar{\hat{y}}_i \sim N(\mu, \frac{\sigma^2}{n})$$

Number of Bootstrap sets

- ntree = 붓스트랩 셋 = 개별 트리 수
- ntree가 클 수록 OOB Error 가 감소하는 것을 볼 수 있습니다.
- 트리의 수가 많을 수록 분산이 작아져 MSE가 감소합니다. 하지만 개별 트리의 수가 지나치게 많을 경우 계산 시간이 증가합니다.

ntree = 300이후부터는 오차가 안정화 되는 것으로 보이므로 default ntree = 500을 그대로 사용하겠습니다.

06 실습 - 3. 배깅 vs 랜덤포레스트 : mtry=p vs mtry=5

```
52 #3. 배깅 vs 랜덤포레스트
53 p<-length(Boston_train)-1 #p=독립변수의 수
54
55 #mtry=p : 배깅
56 set.seed(1234)
57 Boston_rf_p <- randomForest(medv ~ ., data=Boston_train, ntree=500, mtry=p)
58 Boston_rf_p_pred <- predict(Boston_rf_p, Boston_test)
59
60 #랜덤포레스트 / mtry=5
61 set.seed(1234)
62 Boston_rf_5 <- randomForest(medv ~ ., data=Boston_train, ntree=500, mtry=5)
63 Boston_rf_5_pred <- predict(Boston_rf_5, Boston_test)
64
65 MSE(Boston_rf_p_pred, Boston_test$medv)
66 MSE(Boston_rf_5_pred, Boston_test$medv) #MSE 감소 함
```

Console Terminal × Jobs ×

```
~/MyRHome/ ➔
> MSE(Boston_rf_p_pred, Boston_test$medv)
[1] 9.178852
> MSE(Boston_rf_5_pred, Boston_test$medv)
[1] 6.438012
```

mtry = 노드 분할마다 고려되는 속성(변수)의 수

<Bagging vs Randomforest>

- 공통점:
여러 개의 단일 트리를 결합
- 차이점:
랜덤포레스트는 Random subspace 확보를 위해 트리 성장과 정에서 노드마다 고려하는 변수의 수를 한정시킨다.

randomForest()에서 mtry값을 독립변수의 수로 두면 배깅 모델이 됩니다!!

결과: 랜덤포레스트 모형의 MSE가 낮습니다. Why?? [Correlation between trees.](#)

06 실습 - 3. 배깅 vs 랜덤포레스트 : mtry=p vs mtry=5

❑ Correlation between trees : 변수의 개수를 제한하지 않으면 생성되는 개별 트리들은 비슷한 구조를 갖습니다.

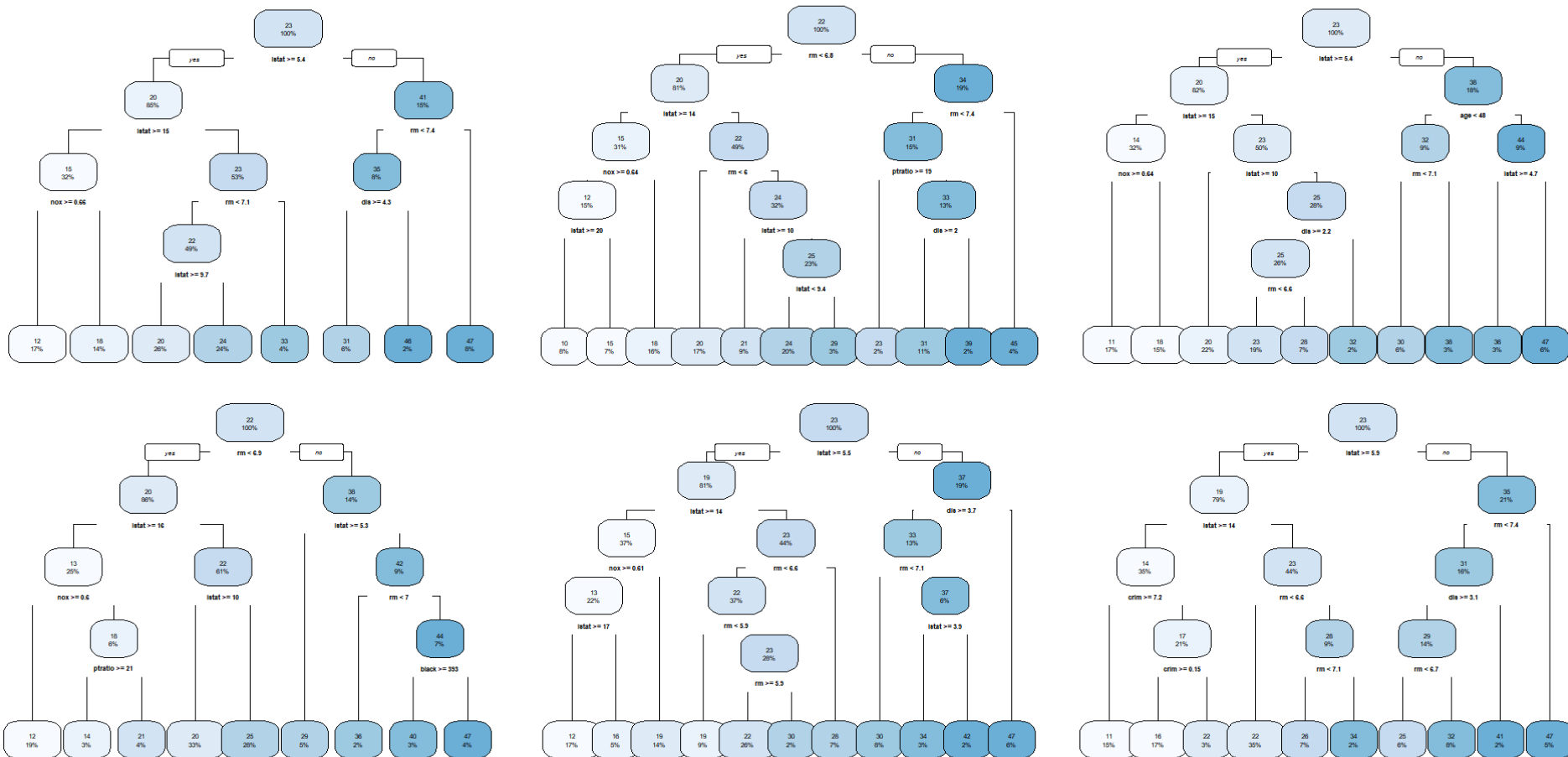
❑ 직접 Bootstrap Sample을 생성해서 각각의 트리를 만들어 볼게요.

```
74 #붓스트랩 셋 6개를 생성하고 6개의 개별트리를 학습시킨다
75 B_reg.rt <- c()
76 for(i in 1:6){
77   B_idx <- sample(1:nrow(Boston_train), replace = T)
78   B_set <- Boston_train[B_idx,]
79   B_reg.rt[[i]] <- rpart(medv ~ ., data=B_set)
80 }
81
82 #각각의 개별트리의 플랏
83 library(rpart.plot)
84 par(mfrow=c(2,3))
85 for(i in 1:6){
86   rpart.plot(B_reg.rt[[i]], cex=0.8)
87 }
88 par(mfrow=c(1,1))
```

Bootstrap Set: 6개
>> 개별 트리 6개

Plot은 다음 슬라이드에!

06 실습 - 3. 배깅 vs 랜덤포레스트 : mtry=p vs mtry=5



- 6개의 트리 상류에서 비슷한 구조를 가짐: 대부분 lstat, rm 속성을 기준으로 분류된다.

비슷한 구조의 트리



비슷한 트리들로 예측



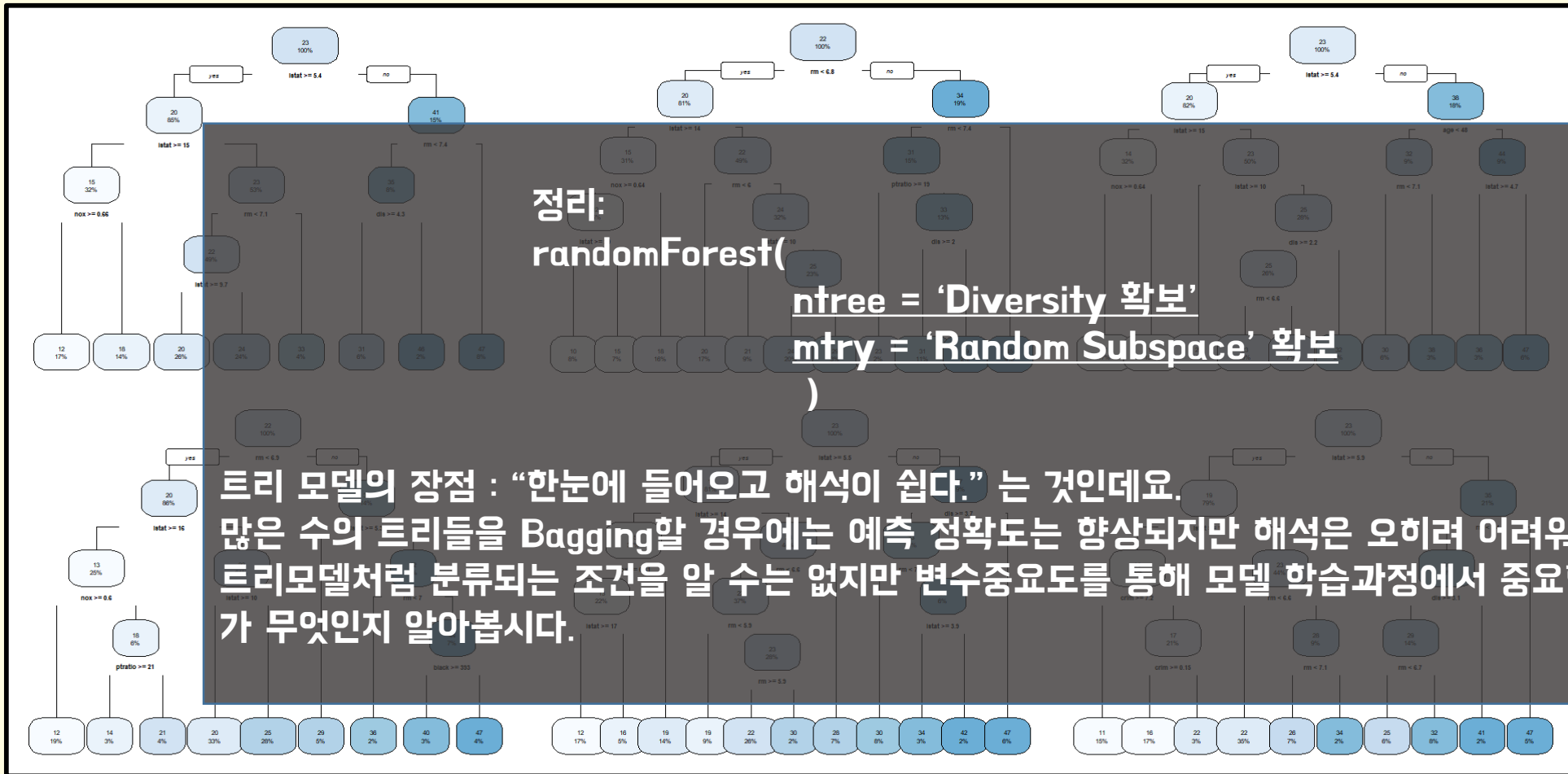
특정 구조에 Overfitting



분산 증가

“ 즉, mtry를 제한하는 이유는 트리 간의 독립성을 높이기 위함. “
위의 과정을 seed 없이 여러번 반복해서 개별트리의 상위 속성이 비슷한지 확인해보세요

06 실습 - 3. 배깅 vs 랜덤포레스트 : mtry=p vs mtry=5



- 6개의 트리 상류에서 비슷한 구조를 가짐: 대부분 lstat, rm 속성을 기준으로 분류된다.

비슷한 구조의 트리



비슷한 트리들로 예측



구조에 Overfitting



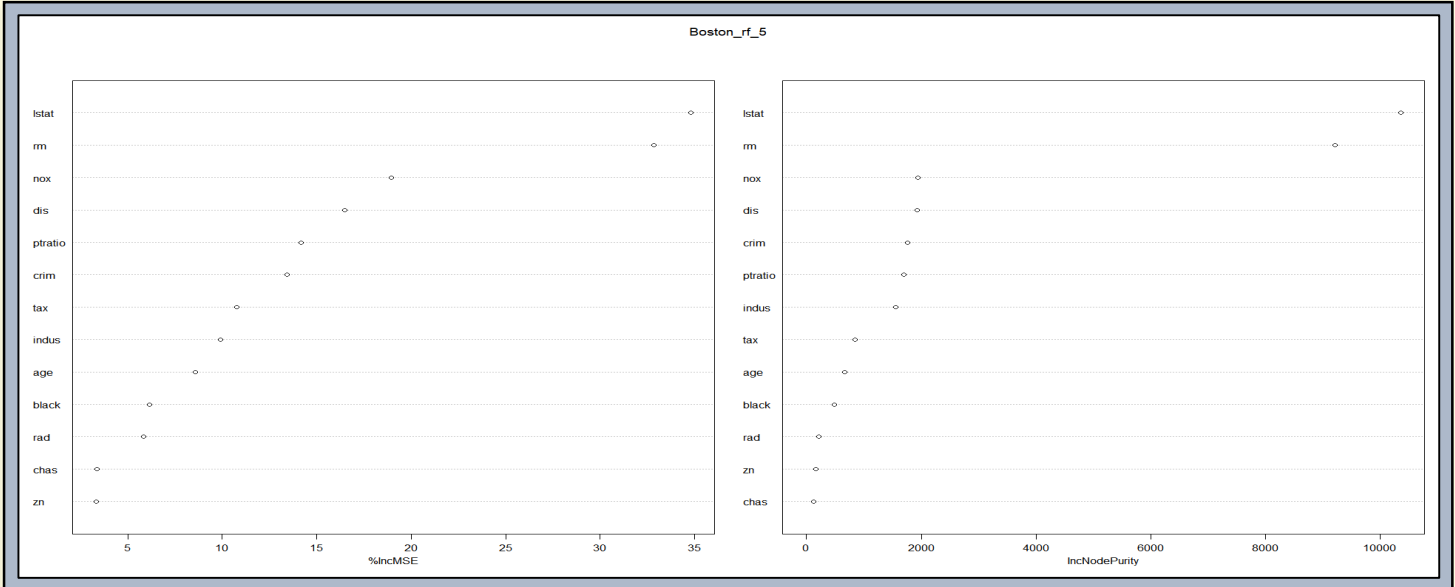
분산 증가

“ 즉, mtry를 제한하는 이유는 트리 간의 독립성을 높이기 위함. “
위의 과정을 seed 없이 여러번 반복해서 개별트리의 상위 속성이 비슷한지 확인해보세요

06 실습 - 4. 변수 중요도

```
실습코드.R* x  Untitled1* x
Source on Save
86 #3. 변수 중요도
87 set.seed(1234)
88 Boston_rf_5 <- randomForest(medv ~ ., data=Boston_train, ntree=500, mtry=5, importance=T)
89 importance(Boston_rf_5)
90 varImpPlot(Boston_rf_5)
91 #%IncMSE: permutation(순서섞기)방법으로 변수중요도 계산
92 #IncNodePurity: MSE의 감소합의 평균, 분류는 gini index
```

```
> importance(Boston_rf_5)
      %IncMSE  IncNodePurity
crim    13.438824    1765.7171
zn       3.350286     162.7929
indus    9.910178    1552.0865
chas     3.376534     118.7433
nox     18.937849    1938.4116
rm      32.821192    9212.5281
age      8.569789     661.1844
dis     16.494676    1926.5094
rad      5.841719     209.2691
tax     10.756543     838.7128
ptratio  14.176798    1695.4473
black    6.148593     478.1767
lstat    34.800764   10361.1250
```

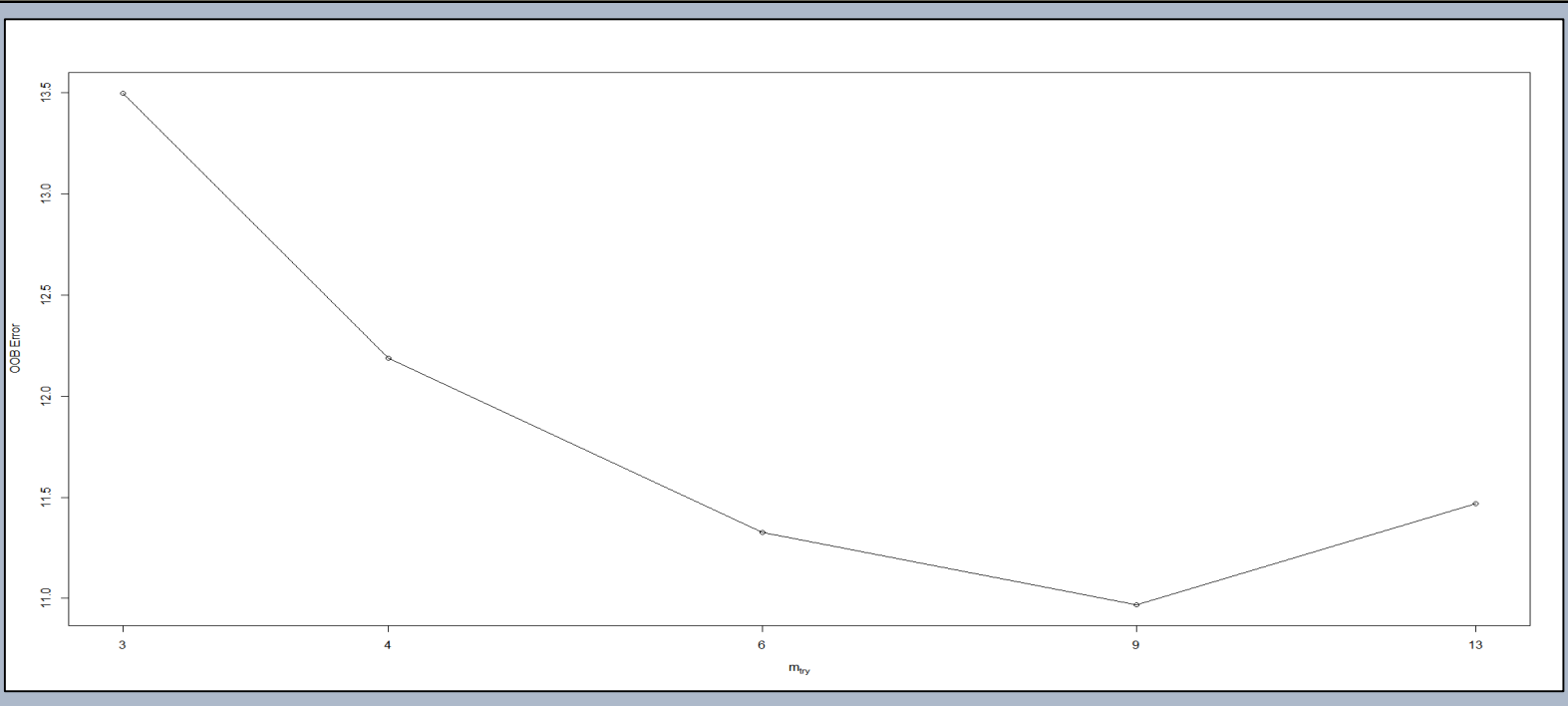


%IncMSE : 해당 변수를 Pumutaion(순서섞기)한 이후 MSE의 증가분

IncNodePurity : 해당 변수를 사용한 노드의 MSE 감소의 합의 평균

06 실습 - 5. Hyperparameter Tuning

```
실습 최종.R x 실습코드.R x
Source on Save
96 #tuneRF
97 features <- setdiff(names(Boston_train), "medv") #medv 변수만 골라내기
98 set.seed(1234)
99 tuneRF(
100   x=Boston_train[features], y=Boston_train$medv,
101   ntreeTry = 500, stepFactor=1.5, improve = 0.01
102 ) #mtry=9에서 OOB Error 최소
```



randomForest 패키지 안에 있는 tuneRF()함수로 mtry를 찾아보겠습니다.

- mtry가 디폴트 값으로 시작해서 stepFactor=1.5씩 곱해지거나 나눠지면서 바뀜
- mtry 값마다 OOB Error계산
- OOB Error가 improve 만큼 개선되면 계속 진행함, 아니면 멈춤

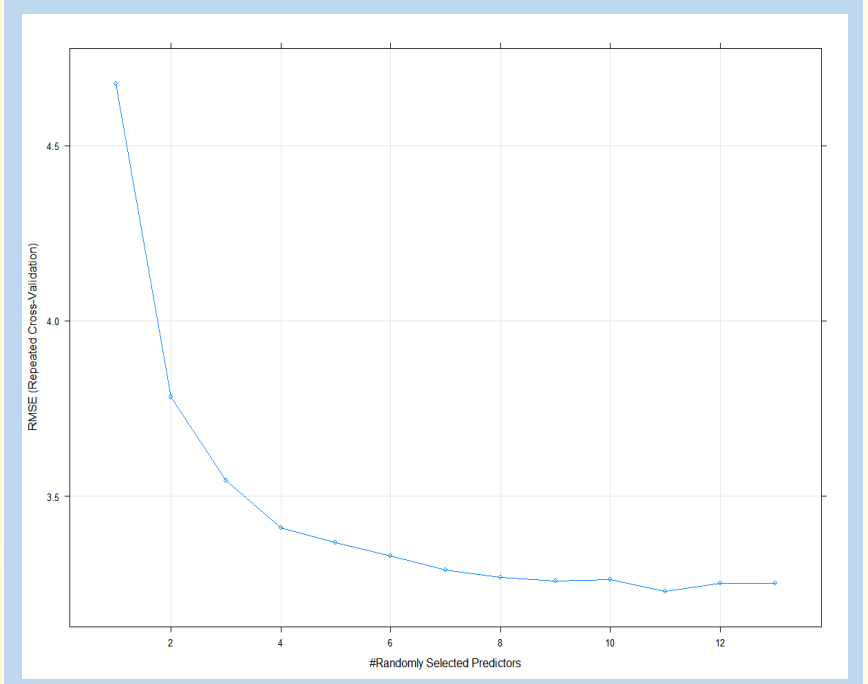
OOB Data가 Validation Set으로 기능하기 때문에 Train Data의 손실을 막을 수 있습니다!

06 실습 - 5. Hyperparameter Tuning

```
실습 최종.R* x 실습코드.R x
Source on Save
104 #caret를 활용한 Hyperparameter 도출
105 fitControl <- trainControl(method = "repeatedcv", number = 10, repeats = 5)
106
107 set.seed(1234)
108 rf_fit <- train(medv ~ ., data = Boston_train, method = "rf",
109               |trControl = fitControl)
110 rf_fit
111 plot(rf_fit) #mtry=13에서 RMSE 최소
112
113 #그리드 서치: 탐색 범위 직접 설정
114 customGrid <- expand.grid(mtry = 1:13)
115 set.seed(1234)
116 rf_fit2 <- train(medv ~ ., data = Boston_train,
117                 method = "rf", trControl = fitControl,
118                 tuneGrid = customGrid)
119 rf_fit2
120 rf_fit2$results[which.min(rf_fit2$results$RMSE),]
121 plot(rf_fit2) #mtry=11에서 RMSE 최소
```

```
실습 최종.R* x 실습코드.R x
Source on Save
124 #MSE를 구해봅시다
125 set.seed(1234)
126 boston_rf_9 <- randomForest(medv ~ ., data = Boston_train, ntree=500, mtry = 9)
127 boston_rf_9_pred <- predict(boston_rf_9, Boston_test)
128
129 boston_rf_fit2_pred <- predict(rf_fit2, Boston_test)
130
131 MSE(Boston_test$medv, boston_rf_9_pred)
132 MSE(Boston_test$medv, boston_rf_fit2_pred)
```

- caret 패키지를 사용하여 CV로 튜닝할 수도 있습니다
- train()실행되는데 조금 오래걸립니다...



```
> MSE(Boston_test$medv, boston_rf_9_pred)
[1] 7.811276
> MSE(Boston_test$medv, boston_rf_fit2_pred)
[1] 8.660696
```

tuneRF로 튜닝한 결과가 성능이 더 좋네요.

06 실습 - 6. range() : 속도개선

- randomForest()의 단점 : 너무 느리다.
- ranger()함수로 더 큰 data set을 다루어 봅시다

```
144 #데이터 셋: AmesHousing 패키지의 make_ames() 셋
145 #train / test 분할
146 library(rsample)
147 set.seed(1234)
148 ames_split <- initial_split(AmesHousing::make_ames(), prop = .7)
149 ames_train <- training(ames_split)
150 ames_test <- testing(ames_split)
151 str(ames_train)
152
153 #ranger 패키지
154 library(ranger)
155 #range() 실행
156 ames_ranger <- ranger(
157   formula = Sale_Price ~ .,
158   data = ames_train,
159   num.trees = 500,
160   mtry = floor(length(features) / 3)
161 )
162 ames_ranger
163 ames_ranger_pred <- predict(ames_ranger, ames_test)
164
165 #궁금하시면 randomForest 패키지와 실제로 속도차이가 있는지 확인해 보세요~!
166 #ames_rf <- randomForest(Sale_Price ~., data=ames_train)
```

```
> str(ames_train)
Classes 'tbl_df', 'tbl' and 'data.frame': 2051 obs. of 81 variables:
 $ MS_SubClass      : Factor w/ 16 levels "One_Story_1946_and_Newer_All_Styles
 $ MS_Zoning        : Factor w/ 7 levels "Floating_Village_Residential",...: 3
 $ Lot_Frontage     : num  141 80 81 93 74 41 43 39 60 0 ...
 $ Lot_Area         : int  31770 11622 14267 11160 13830 4920 5005 5389 7500 7
 $ Street           : Factor w/ 2 levels "Grvl","Pave": 2 2 2 2 2 2 2 2 2 ..
```

```
Console Terminal Jobs
~/MyHome/ ↵
> ames_ranger
Ranger result

Call:
ranger(formula = Sale_Price ~ ., data = ames_train, num.trees = 500, mtry = floor(length(features)/3))

Type: Regression
Number of trees: 500
Sample size: 2051
Number of independent variables: 80
Mtry: 4
Target node size: 5
Variable importance mode: none
Splitrule: variance
OOB prediction error (MSE): 786377161
R squared (OOB): 0.8823306
```

속도가 빠릅니다!!
문법은 randomForest와 매우 유사합니다.
(Hyperparameter 명칭이 조금 다른 것들이 있습니다!!)

06 실습 - 6. range() : Tuning

□ 다양한 하이퍼파라미터 조합으로 해보겠습니다.

1

```
실습 최종.R* x 실습코드.R* x
Source on Save
168 #속도가 향상되었으니 이전보다 다양한 파라미터 조합으로 그리드(HP조합) 생성
169 hyper_grid <- expand.grid(
170   mtry = seq(20, 30, by = 2),
171   num.trees = c(300, 500, 700, 1000),
172   node_size = seq(3, 9, by = 2)
173 )
174 head(hyper_grid)
175 tail(hyper_grid)
176
177 for(i in 1:nrow(hyper_grid)) {
178   # train model
179   model <- ranger(
180     formula = Sale_Price ~ .,
181     data = ames_train,
182     num.trees = hyper_grid$num.trees[i],
183     mtry = hyper_grid$mtry[i],
184     min.node.size = hyper_grid$node_size[i],
185     seed = 1234
186   )
187 }
188
189 # add OOB error to grid
190 hyper_grid$OOB_RMSE[i] <- sqrt(model$prediction.error)
191 } #...조금 오래 걸립니다
192
193 #RMSE(OOB Error)가 가장 낮은 Hyperparameter 조합
194 G <- hyper_grid[which.min(hyper_grid$OOB_RMSE),]; G
> #RMSE(OOB Error)가 가장 낮은 Hyperparameter 조합
> G <- hyper_grid[which.min(hyper_grid$OOB_RMSE),]; G
mtry num.trees node_size OOB_RMSE
6 30 300 3 25526.01
```

2

3

```
실습 최종.R* x 실습코드.R* x
Source on Save
196 #하이퍼파라미터 넣어서 다시 fitting
197 optimal_ranger <- ranger(
198   formula = Sale_Price ~ .,
199   data = ames_train,
200   num.trees = G$num.trees,
201   mtry = G$mtry,
202   min.node.size = G$node_size,
203   importance = 'permutation'
204 )
205 optimal_ranger_pred <- predict(optimal_ranger, ames_test)
206
207 #MSE 감소 확인
208 MSE(ames_ranger_pred$predictions, ames_test$Sale_Price)
209 MSE(optimal_ranger_pred$predictions, ames_test$Sale_Price)
```

4

```
#MSE 감소 확인
> MSE(ames_ranger_pred$predictions, ames_test$Sale_Price)
[1] 620053126
> MSE(optimal_ranger_pred$predictions, ames_test$Sale_Price)
[1] 537564227
```

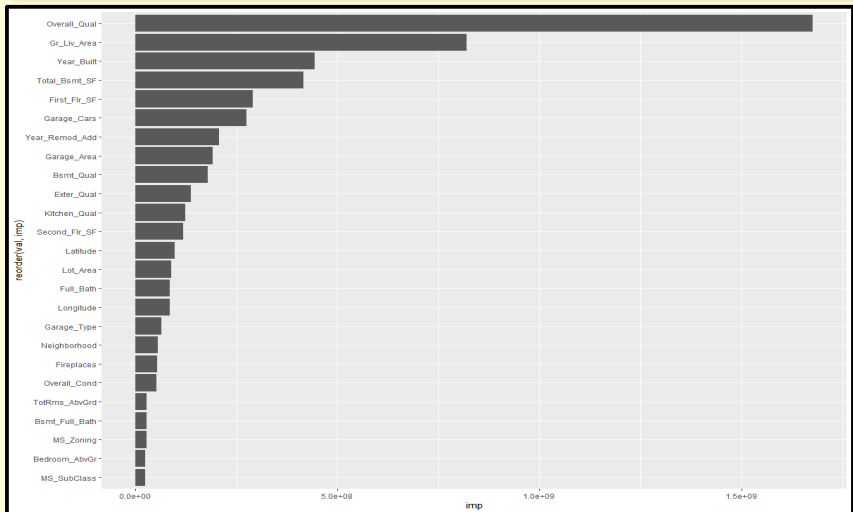
- **ranger** - ranger comes to the rescue when you have high dimensional data and want a memory efficient yet fast implementation of randomForest. The word ranger came from RANdom forest GEnerator. The main purpose where I have used ranger is to build models quickly and find out optimal parameter values using parameter tuning

06 실습 - 6. range() : 변수 중요도

□ randomForest() 와 마찬가지로 변수 중요도를 확인할 수 있습니다.

```
실습 최종.R* x 실습코드.R* x
Source on Save
212 #변수 중요도
213 optimal_ranger$variable.importance
214
215 library(dplyr)
216 #변수 중요도 플랏
217 imp <- as.vector(optimal_ranger$variable.importance)
218 val <- rownames(data.frame(optimal_ranger$variable.importance))
219 valimp <- data.frame(val, imp)
220 valimp_tidy <- valimp %>% arrange(desc(imp)) %>% top_n(25)
221 valimp_tidy
222 ggplot(valimp_tidy, aes(reorder(val, imp), imp))+geom_col()+coord_flip()
```

```
Console Terminal x Jobs x
~/MyRHome/
> valimp_tidy
      val      imp
1 Overall_Qual 1676730466
2 Gr_Liv_Area  820016223
3 Year_Built  444151685
4 Total_Bsmt_SF 416537232
5 First_Flr_SF 290056889
6 Garage_Cars  274008481
7 Year_Remod_Add 207689267
8 Garage_Area  190559954
9 Bsmt_Qual    178778616
10 Exter_Qual  136430126
```



07 출처

본 발표 자료는
고려대학교 산업경영공학부 김성범 교수님의 '[핵심 머신러닝] 랜덤포레스트 모델' 강의를
인용하여 만들었습니다.

<https://www.youtube.com/watch?v=llT5-piVtRw&t=700s>

