Gradient Boosting (for regression) : 학습 과정

- Gradient boosting은 boosting 계열에 속하는 것으로 여러 개의 약한 분류기로 데이터를 학습하여 결과를 종합하는 알고리즘이다.
- Gradient boosting은 regression, classification에 모두 적용 가능하며, target (label) 데이터의 residual (잔차)을 학습하는 방식이다.

feature (X) target (Y)

height	color	gender	weight	residual(0)	residual(1)
1.6	B	M	88	16.8	15.1
1.6	G	F	76	4.8	4.3
1.5	B	F	56	-15.2	-13.7
1.8	R	M	73	1.8	1.4
1.5	G	M	77	5.8	5.4
1.4	B	F	57	-14.2	-12.7

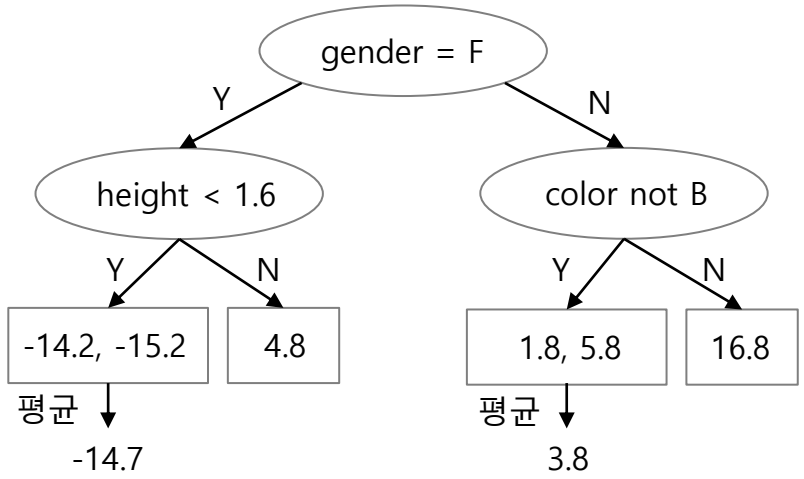
학습 데이터

residual (1)이 더 작아졌다.

1) weight의 평균을 계산한다. 평균 =71.2

2) residual(0)를 계산한다 : weight - 평균
(88 - 71.2 = 16.8)

3) residual (0)에 대해 Decision tree (1)을 생성한다.



4) Tree를 이용해서 새로운 residual (1)을 계산한다 : weight - (평균 + 학습률 * Tree의 leaf 평균)

$$88 - (71.2 + 0.1 * 16.8) = 15.1$$
$$76 - (71.2 + 0.1 * 4.8) = 4.3$$
$$56 - (71.2 - 0.1 * 14.7) = -13.7$$
$$73 - (71.2 + 0.1 * 3.8) = 1.4$$
$$77 - (71.2 + 0.1 * 3.8) = 5.4$$
$$57 - (71.2 - 0.1 * 14.7) = -12.7$$

Gradient Boosting (for regression) : 학습 과정

- Residual이 더 이상 줄어들지 않을 때까지 이 과정을 반복한다.

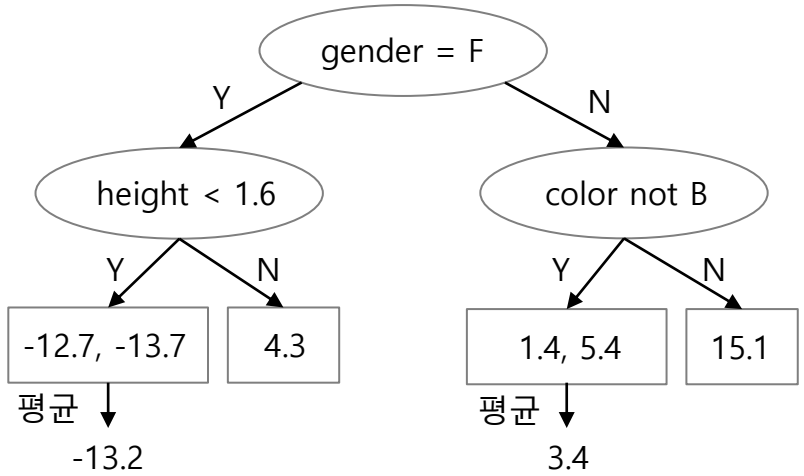
feature (X) target (Y)

height	color	gender	weight	residual(0)	residual(1)	residual(2)
1.6	B	M	88	16.8	15.1	13.6
1.6	G	F	76	4.8	4.3	3.9
1.5	B	F	56	-15.2	-13.7	-12.4
1.8	R	M	73	1.8	1.4	1.1
1.5	G	M	77	5.8	5.4	5.1
1.4	B	F	57	-14.2	-12.7	-11.4

학습 데이터

residual이 점점 작아지고 있다.

5) residual (1)에 대해 Decision tree (2)를 생성한다.



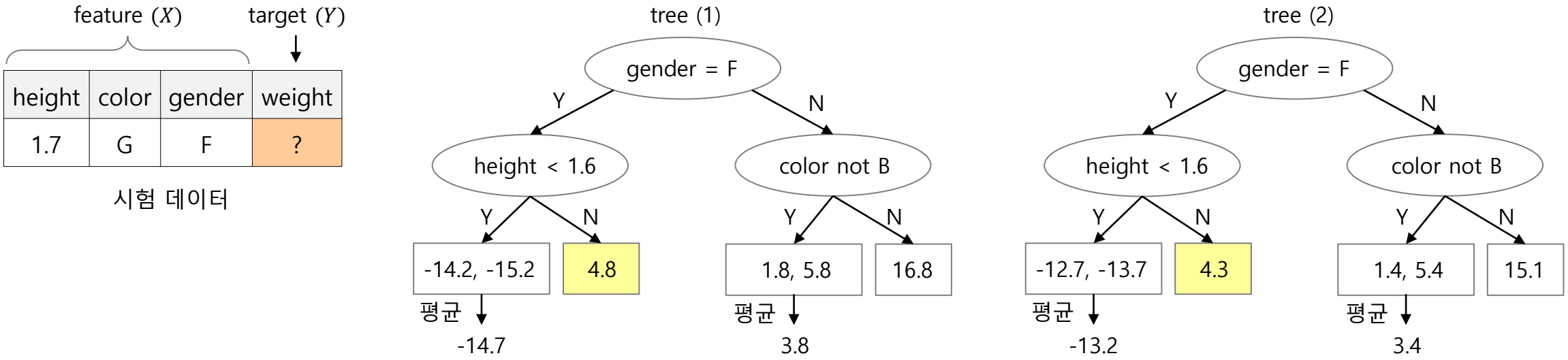
실제는 트리의 분기가 tree(1)과는 다를 것이다.

6) Tree (1)과 tree (2)를 이용해서 새로운 residual (2)를 계산한다 : weight - (평균 + 학습률 * tree(1)의 leaf 평균 + 학습률 * tree(2)의 leaf 평균)

$$88 - (71.2 + 0.1 * 16.8 + 0.1 * 15.1) = 13.6$$
$$76 - (71.2 + 0.1 * 4.8 + 0.1 * 4.3) = 3.9$$
$$56 - (71.2 - 0.1 * 14.7 - 0.1 * 13.2) = -12.4$$
$$73 - (71.2 + 0.1 * 3.8 + 0.1 * 3.4) = 1.1$$
$$77 - (71.2 + 0.1 * 3.8 + 0.1 * 3.4) = 5.1$$
$$57 - (71.2 - 0.1 * 14.7 - 0.1 * 13.2) = -11.4$$

Gradient Boosting (for regression) : 추정 과정

- 학습이 완료된 여러 개의 서브-트리들을 이용하여 시험 데이터의 추정치를 구하는 방법은 아래와 같다.



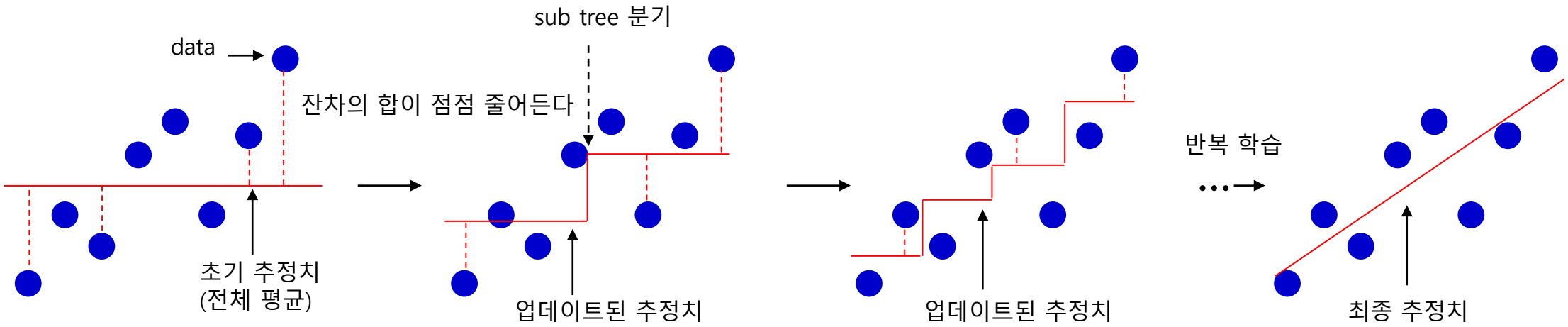
7) Tree (1)과 tree (2)를 이용해서 시험 데이터의 weight를 추정한다.

$$71.2 + \underbrace{0.1 * 4.8}_{\text{tree (1)}} + \underbrace{0.1 * 4.3}_{\text{tree (2)}} = 72.1$$

- 여기서는 단순한 예시를 위해 residual을 3회만 계산했다.
- 실제는 반복 횟수를 더 증가 시켜야 한다.
- 실제는 tree(1)과 tree(2)가 다르게 분기할 것이다.

Gradient Boosting (for regression) : 학습 과정의 직관적 이해

- 모든 데이터의 평균으로 초기 추정치를 대충 설정한 다음, 잔차를 계산한다. (잔차 = 데이터 - 초기 추정치)
- Decision tree를 생성해서 잔차를 줄이는 방향으로 학습하고, 추정치를 업데이트하고, 새로운 잔차를 계산한다.
- 새로운 decision tree를 생성해서 새로운 잔차를 학습해서 잔차를 계속 줄여 나가고, 추정치도 계속 업데이트 한다.
- 잔차가 더 이상 줄어들지 않는 수준까지 decision tree를 생성하면서, 추정치를 업데이트 한다.
- 최종 추정치는 아래 그림처럼 점차 데이터를 잘 설명할 수 있는 상태가 된다.



Gradient Boosting (for regression) : 알고리즘 해석

Input: Data $\{(x_i, y_i)\}_{i=1}^{n}$, and a differentiable loss function $L(y_i, F(x))$

Step 1: Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

Step 2: for m = 1 to M:

(A) Compute so-called pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad for\ i = 1, \dots, n$$

(B) Fit a regression tree to the r_{im} values and create terminal regions R_{jm}, $for\ j = 1, \dots, J_m$

(C) $for\ j = 1, \dots, J_m$ compute

$$\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{ij}} L(y_i, F_{m-1}(x_i) + \gamma)$$

(D) Update the model:

$$F_m(x) = F_{m-1}(x) + \alpha \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$$

3. Output $F_M(x)$

$$y = \{88, 76, 56, 73, 77, 57\} \qquad L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial}{\partial \gamma}\left(\frac{1}{2}(88-\gamma)^2 + \frac{1}{2}(76-\gamma)^2 \dots \frac{1}{2}(57-\gamma)^2\right) = 0$$

$$-(88-\gamma) - (76-\gamma) - \cdots - (57-\gamma) = 0$$

$$\gamma = \frac{88+76+56+73+77+57}{6} = 71.2 = F_0(x)$$

* 초기 추정치는 모두 평균값인 71.2 이다.

m=1 : 첫 번째 반복

$$r_{1,1} = -\frac{\partial L(y_1, F_0(x_1))}{\partial F_0(x_1)} = -\frac{1}{2}\frac{\partial}{\partial F_0(x_1)}(y_1 - F_0(x_1))^2$$

$$= y_1 - F_0(x_1) = 88 - 71.2 = 16.8 \quad \leftarrow \text{ i=1 : 첫 번째 데이터}$$

$$r_{i,1} = \{16.8, 4.3, -13.7, 1.4, 5.4\} \quad \leftarrow \text{ 모든 데이터의 residual}$$

Gradient Boosting (for regression) : 알고리즘 해석

Input: Data $\{(x_i, y_i)\}_{i=1}^{n}$, and a differentiable loss function $L(y_i, F(x))$

Step 1: Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

Step 2: for m = 1 to M:

(A) Compute so-called pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad for\ i = 1, \dots, n$$

(B) Fit a regression tree to the r_{im} values and create terminal regions R_{jm}, $for\ j = 1, \dots, J_m$

(C) $for\ j = 1, \dots, J_m$ compute

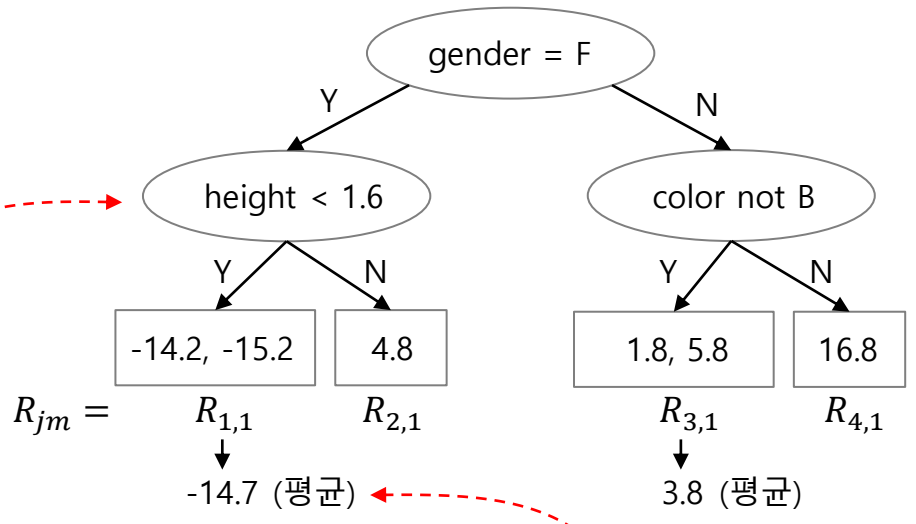
$$\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{ij}} L(y_i, F_{m-1}(x_i) + \gamma)$$

(D) Update the model:

$$F_m(x) = F_{m-1}(x) + \alpha \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$$

3. Output $F_M(x)$

tree (1) : m = 1



$$\gamma_{1,1} \rightarrow \frac{1}{2}\frac{\partial}{\partial \gamma}[(57 - 71.2 - \gamma)^2 + (56 - 71.2 - \gamma)^2] = 0$$

$$\frac{1}{2}\frac{\partial}{\partial \gamma}[(-14.2 - \gamma)^2 + (-15.2 - \gamma)^2] = 0$$

$$14.2 + \gamma + 15.2 + \gamma = 0 \ \rightarrow \ \gamma = \frac{-14.2 - 15.2}{2} = -14.7 = \gamma_{1,1}$$

$$F_1(x_1) = F_0(x_1) + 0.1 * R_{4,1} = 71.2 + 0.1 * 16.8 = 72.88$$

* 첫 번째 데이터 (i=1)의 추정치 = 72.88 ($R_{4,1}$: 첫 번째 데이터의 leaf node 값)

Gradient Boosting : Regression 연습

* 실습 파일 : 3-5.GradientBoost(regression).py

- Gradient Boosting (regressor) 로 여러 개의 feature를 가진 Boston housing 데이터를 학습하고 성능을 평가한다.

```
# Gradient Boosting Machine (GBM)으로 Boston Housing 데이터를 학습한다.
# GBM for regression
# ------------------------------------------------------------------
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

# Boston housing data set을 읽어온다
boston = load_boston()

# Train 데이터 세트와 Test 데이터 세트를 구성한다
x = boston.data
y = boston.target
trainX, testX, trainY, testY = train_test_split(x, y, test_size = 0.2)

# Gradient Boosting (regressor)로 Train 데이터 세트를 학습한다.
# default:
# loss = least square
# learning_rate = 0.1
# n_estimators = 100
# max_depth = 3
model = GradientBoostingRegressor(loss='ls', learning_rate=0.1,
                                  n_estimators=100, max_depth=3)
model.fit(trainX, trainY)

# testX[n]에 해당하는 target (price)을 추정한다.
n = 1
price = model.predict(testX[n].reshape(1,-1))
```

```
print('test[%d]의 추정 price = %.2f' % (n, price))
print('test[%d]의 실제 price = %.2f' % (n, testY[n]))
print('추정 오류 = rmse(추정 price - 실제 price) = %.2f' %
      np.sqrt(np.square(price - testY[n])))

# 시험 데이터 전체의 오류를 MSE로 표시한다.
# MSE는 값의 범위가 크다는 단점이 있다.
predY = model.predict(testX)
rmse = (np.sqrt(mean_squared_error(testY, predY)))
print('\n시험 데이터 전체 오류 (rmse) = %.4f' % rmse)

# 시험 데이터 전체의 오류를 R-square로 표시한다.
print('\n시험 데이터 전체 오류 (R2-score) = %.4f' % model.score(testX, testY))
```

결과 :

```
test[1]의 추정 price = 19.22
test[1]의 실제 price = 23.20
추정 오류 = rmse(추정 price - 실제 price) = 3.98

시험 데이터 전체 오류 (rmse) = 2.8167

시험 데이터 전체 오류 (R2-score) = 0.8945
```

Gradient Boosting (for classification) : 학습 과정

- Gradient Boosting 알고리즘으로 classification을 수행하는 과정은 아래와 같다. 전체 개념은 regression과 동일하나, 추정치를 위해 odds, log(odds), probability 개념을 사용하고, loss 함수로는 binary cross entropy를 사용한다.

feature (X) target (Y)

Likes Popcorn	Age	Color	Loves Troll 2	Residual (0)	Prob	Residual (1)
Yes	12	B	Yes	0.3	0.9	0.1
Yes	87	G	Yes	0.3	0.5	0.5
No	44	B	No	-0.7	0.5	-0.5
Yes	19	R	No	-0.7	0.1	-0.1
No	32	G	Yes	0.3	0.9	0.1
No	14	B	Yes	0.3	0.9	0.1

1) target의 $\log(odds)$를 계산한다. $\log\left(\frac{p(Yes)}{1-p(Yes)}\right) = \log\left(\frac{4}{2}\right) = 0.69$

2) $P(Y = Yes)$를 계산한다.

$$P(Y = Yes) = \frac{1}{1 + \exp(-log(odds))} = 0.67 \cong 0.7$$

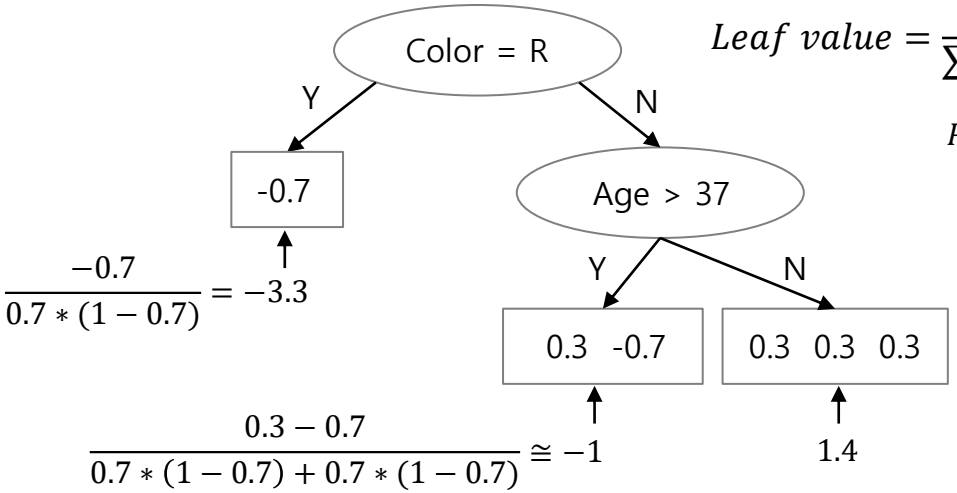
3) Residual (0)을 계산한다.

첫 번째 데이터 : $1 - 0.7 = 0.3 \quad (Yes = 1, No = 0)$

4) residual (0)에 대해 Decision tree (1)을 생성하고 아래 식으로 leaf-value를 계산한다.

$$Leaf\ value = \frac{\sum_i Resitual_i}{\sum_i [P_i * (1 - P_i)]}$$

P_i : 이전 확률



5) 각 데이터의 probability를 계산하고, Residual (1)을 계산한다 (학습률 = 0.8 적용).

첫 번째 데이터 : log(odds) $prediction = log\left(\frac{4}{2}\right) + 0.8 * 1.4 = 1.8$

$$P(Y = Yes) = \frac{1}{1 + \exp(-1.8)} = 0.9$$

$$Residual\ (1)_1 = 1 - 0.9 = 0.1$$

Gradient Boosting (for classification) : 학습 및 시험 데이터 추정 과정

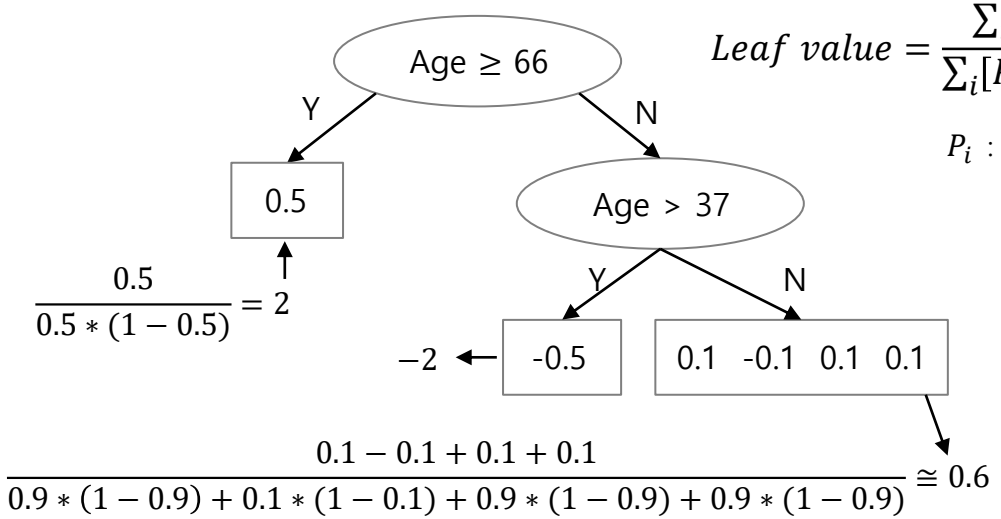
feature (X) / target (Y)

Likes Popcorn	Age	Color	Loves Troll 2	Residual (0)	Prob	Residual (1)
Yes	12	B	Yes	0.3	0.9	0.1
Yes	87	G	Yes	0.3	0.5	0.5
No	44	B	No	-0.7	0.5	-0.5
Yes	19	R	No	-0.7	0.1	-0.1
No	32	G	Yes	0.3	0.9	0.1
No	14	B	Yes	0.3	0.9	0.1

6) residual (1)에 대해 Decision tree (2)를 생성하고 아래 식으로 leaf-value를 계산한다.

$$Leaf\ value = \frac{\sum_i Resitual_i}{\sum_i [P_i * (1 - P_i)]}$$

P_i : 이전 확률



7) 각 데이터의 probability를 계산하고, Residual (2)를 계산한다 (학습률 = 0.8 적용).

첫 번째 데이터 : log(odds) $prediction = log\left(\frac{4}{2}\right) + 0.8 * 1.4 + 0.8 * 0.6 = 2.3$

$$P(Y = Yes) = \frac{1}{1 + \exp(-2.3)} = 0.91$$

8) Residual이 더 이상 줄어들지 않을 때까지 반복한다.

시험 데이터

Likes Popcorn	Age	Color	Loves Troll 2
Yes	25	G	?

← Yes

log(odds) $prediction = log\left(\frac{4}{2}\right) + 0.8 * 1.4 + 0.8 * 0.6 = 2.3$

$P(Y = Yes) = \frac{1}{1 + \exp(-2.3)} = 0.91$ ← Yes

Gradient Boosting (for classification) : 알고리즘 해석

Input: Data $\{(x_i, y_i)\}_{i=1}^{n}$, and a differentiable loss function $L(y_i, F(x))$

Step 1: Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

Step 2: for m = 1 to M:

(A) Compute so-called pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad for\ i = 1, \dots, n$$

(B) Fit a regression tree to the r_{im} values and create terminal regions R_{jm}, $for\ j = 1, \dots, J_m$

(C) $for\ j = 1, \dots, J_m$ compute

$$\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{ij}} L(y_i, F_{m-1}(x_i) + \gamma)$$

이 부분은 Taylor 급수를 이용해서 근사적으로 계산해야 함. (생략)

(D) Update the model:

$$F_m(x) = F_{m-1}(x) + \alpha \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$$

3. Output $F_M(x)$

$$L = -\sum_{i=1}^{n} [y_i \log(p) + (1 - y_i)\log(1 - p)]$$ ← cross entropy or negative log likelihood

↓ 유도 과정 생략

$$L = -\sum_{i=1}^{n} [y_i \log(odds) + \log(1 + \exp(\log(odds)))]$$

$$\frac{\partial L}{\partial \log(odds)} = -\sum_{i=1}^{n}\left[y_i - \frac{1}{1 + \exp(-\log(odds))}\right] = -\sum_{i=1}^{n}(y_i - p_i)$$

$$= -(1 - p + 1 - p + 0 - p + 0 - p + 1 - p + 1 - p) = 0 \qquad p = \frac{4}{6} = 0.67 \cong 0.7$$

$$\log(odds) = log\left(\frac{p}{1-p}\right) = log\left(\frac{0.67}{0.33}\right) = log\left(\frac{4}{2}\right) = F_0(x) \cong 0.7$$

Yes=4개, No=2개의 의미 / 6개 중에 Yes=4개의 의미

m=1 : 첫 번째 반복

i=1 : 첫 번째 데이터의 residual

$$r_{1,1} = -\frac{\partial L(y_1, F_0(x_1))}{\partial F_0(x_1)} = y_1 - \frac{1}{1 + \exp(-F_0(x_1))} = 1 - 0.67 \cong 0.3$$

$$r_{i,1} = \{0.3, 0.3, -0.7, -0.7, 0.3, 0.3\}$$ ← 모든 데이터의 residual

Gradient Boosting : Classification 연습

* 실습 파일 : 3-6.GradientBoost(classification).py

- Gradient Boosting (classifier) 로 Breast Cancer 데이터를 학습하고 성능을 평가한다.

```
# Gradient Boosting Machine (GBM)으로 Breast Cancer 데이터를 학습한다.
# GBM for classification
# ------------------------------------------------------------------
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier

# breast cancer 데이터를 가져온다.
cancer = load_breast_cancer()

# Train 데이터 세트와 Test 데이터 세트를 구성한다
trainX, testX, trainY, testY = \
    train_test_split(cancer['data'], cancer['target'], test_size = 0.2)

# Gradient Boosting (classifier)로 Train 데이터 세트를 학습한다.
# default:
# loss = deviance (=logistic regression)
# learning_rate = 0.1
# n_estimators = 100
# max_depth = 3
model = GradientBoostingClassifier(loss='deviance', learning_rate=0.1,
                                   n_estimators=100, max_depth=3)
model.fit(trainX, trainY)

# Test 세트의 Feature에 대한 class를 추정하고, 정확도를 계산한다
# accuracy = model.score(testX, testY)와 동일함.
predY = model.predict(testX)
accuracy = (testY == predY).mean()
print()
print("* 시험용 데이터로 측정한 정확도 = %.2f" % accuracy)
```

```
# Train 세트의 Feature에 대한 class를 추정하고, 정확도를 계산한다
predY = model.predict(trainX)
accuracy = (trainY == predY).mean()
print("* 학습용 데이터로 측정한 정확도 = %.2f" % accuracy)


결과 :

* 시험용 데이터로 측정한 정확도 = 0.96
* 학습용 데이터로 측정한 정확도 = 1.00
```

XGBoost (Extreme Gradient Boosting) - (for regression) : 학습 과정

- Gradient Boosting은 서브-트리의 leaf node 값의 평균 값을 이용하여 residual을 줄여 나갔으나, XGBoost는 loss 함수가 최소가 되는 최적값 (optimal output value)을 찾아 residual을 줄여 나간다. XGBoost는 GBM보다 메모리 효율이 좋고 속도도 빠르다. 대용량 데이터 처리에 성능이 우수하다.
- 또한 XGBoost는 regularization과 pruning을 통해 overfitting을 줄이고 일반화 특성을 좋게 만든다.

feature (X) target (Y)

Drug dosage	Drug effect	Residual (0)
13	-10	-10.5
21	7	6.5
28	8	7.5
32	-7	-7.5

1) 초기 추정치를 임의로 설정하고 (ex : 0.5)

residual (0)을 계산한다.

residual (0) = -10 - 0.5 = -10.5
7 - 0.5 = 6.5
8 - 0.5 =7.5
-7 - 0.5 = -7.5

2) residual (0)에 대해 similarity와 output value를 계산한다.

-10.5, 6.5, 7.5, -7.5 $\lambda = 1$을 적용했음.

$$similarity = \frac{(-10.5 + 6.5 + 7.5 - 7.5)^2}{4 + 1} = 3.2$$

$$output = \frac{-10.5 + 6.5 + 7.5 - 7.5}{4 + 1} = -0.8$$

$$L = \sum_{i=1}^{n} L(y_i, p_i) + \gamma T + \frac{1}{2}\lambda O^2$$

($L(y_i, p_i)$ ↑ mse)

- T : 터미널 노드 개수
- O : output value
- λ : regularization constant
- γ : pruning 판단 상수

↓ minimize w.r.t output value (γ는 생략했음)

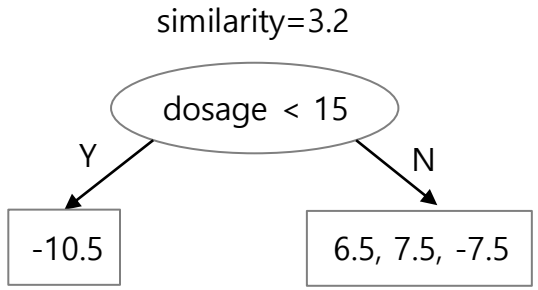
$$output\ value(O) = \frac{\sum_{i=1}^{n} resudual}{|residual| + \lambda}$$

$$similarity\ score = \frac{(\sum_{i=1}^{n} resudual_i)^2}{|residual| + \lambda}$$

Loss가 최소가 되는 Tree의 조건

$|residual|$ ← residual 개수

3) residual (0)에 대해 tree를 생성한다.



similarity=55.1 similarity=10.6

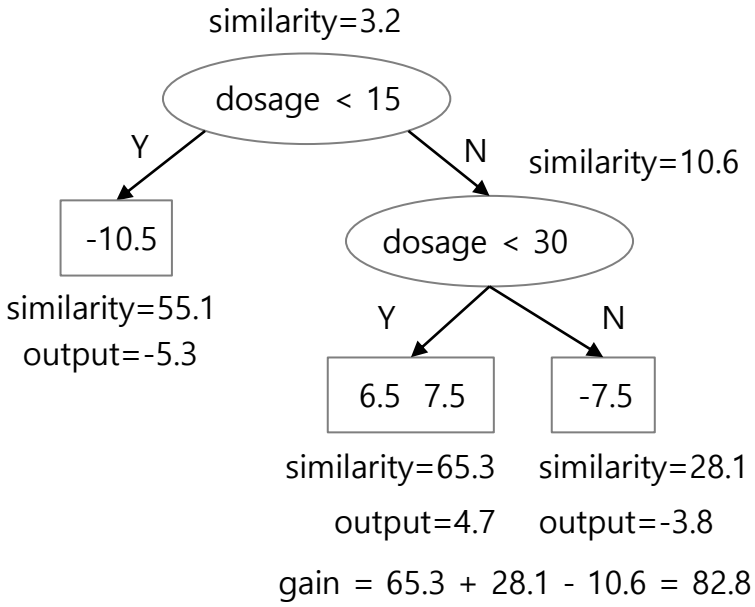
gain = 55.1 + 10.6 - 3.2 = 62.5

분할로 인해 62.5 만큼 이득이 생겼음.

- 각 leaf 노드의 similarity score를 계산한다.
- Gain이 가장 큰 분기를 찾아 tree를 생성한다. (weighted quantile search 방법)
- 이 경우는 dosage < 15 조건일 때의 gain이 가장 크다. Loss도 작아진다.
- leaf 노드에 residual이 유사한 것끼리 모여 있으면 similarity score가 높다. 상쇄되는 부분이 작기 때문이다.

XGBoost (Extreme Gradient Boosting) - (for regression) : 학습 과정

4) tree를 완성하고 leaf 노드의 output value를 계산한다.



gain = 65.3 + 28.1 - 10.6 = 82.8

- (6.5, 7.5, -7.5) 노드를 gain이 가장 큰 조건으로 분기한다. (weighted quantile search)
- 이 경우는 dosage < 30 조건일 때의 gain이 가장 크다.
- (6.5, 7.5) 노드는 분기해도 더 이상 이득이 없다 (gain < 0)
- Tree를 분기한 후 gain < γ라면 분기하지 않는다 (pruning)
- 예를 들어 γ = 100이라면 우측 서브 트리는 82.8 < 100이므로 (6.5, 7.5, -7.5) 노드는 분리하지 않는다.
- γ는 분석자가 지정할 하이퍼 파라메터이다.

5) 좌측 tree의 output value를 이용하여 residual (1)을 계산한다.

- 첫 번째 데이터의 추정치 = 0.5 + 0.3 * (-5.3) = -1.09 ← 학습률 (ε) = 0.3
  new residual = -10 - (-1.09) = -8.91
- 두 번째 데이터의 추정치 = 0.5 + 0.3 * 4.7 = 1.91
  new residual = 7 - 1.91 = 5.09
- 세 번째 데이터의 추정치 = 0.5 + 0.3 * 4.7 = 1.91
  new residual = 8 - 1.91 = 6.09
- 네 번째 데이터의 추정치 = 0.5 + 0.3 * (-3.8) = -0.64
  new residual = -7 - (-0.64) = -6.36

Drug dosage	Drug effect	Residual (0)	Residual (1)
13	-10	-10.5	-8.91
21	7	6.5	5.09
28	8	7.5	6.09
32	-7	-7.5	-6.36

← residual이 점점 작아지고 있다. = 추정치가 점점 정확해 지고 있다.

6) residual (1)으로 또 tree를 생성하고, residual이 줄어들지 않을 때까지 반복한다.

- 첫 번째 데이터의 추정치 = 초기 추정치 + ε * 이전 tree의 output + ε * 현재 tree의 output + ...

XGBoost : Regression 연습

* 실습 파일 : 3-7.XGBoost(regression).py

- XGBoost (regressor)로 Boston housing 데이터를 학습하고 성능을 평가한다.

```
# XGBoost로 Boston Housing 데이터를 학습한다.
# XGBoost for regression
# xgboost 설치 : conda install -c anaconda py-xgboost
# ------------------------------------------------
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from xgboost import XGBRegressor

# Boston housing data set을 읽어온다
boston = load_boston()

# Train 데이터 세트와 Test 데이터 세트를 구성한다
x = boston.data
y = boston.target
trainX, testX, trainY, testY = train_test_split(x, y, test_size = 0.2)

# XGBoost (regressor)로 Train 데이터 세트를 학습한다.
model = XGBRegressor(objective='reg:squarederror')  # default로 학습
model.fit(trainX, trainY)

# testX[n]에 해당하는 target (price)을 추정한다.
n = 1
price = model.predict(testX[n].reshape(1,-1))
print('test[%d]의 추정 price = %.2f' % (n, price))
print('test[%d]의 실제 price = %.2f' % (n, testY[n]))
print('추정 오류 = rmse(추정 price - 실제 price) = %.2f' % np.sqrt(np.square(pr

# 시험 데이터 전체의 오류를 MSE로 표시한다.
```

```
# MSE는 값의 범위가 크다는 단점이 있다.
predY = model.predict(testX)
rmse = (np.sqrt(mean_squared_error(testY, predY)))
print('\n시험 데이터 전체 오류 (rmse) = %.4f' % rmse)

# 시험 데이터 전체의 오류를 R-square로 표시한다.
print('\n시험 데이터 전체 오류 (R2-score) = %.4f' % model.score(testX, testY))

결과 :

test[1]의 추정 price = 20.11
test[1]의 실제 price = 18.20
추정 오류 = rmse(추정 price - 실제 price) = 1.91

시험 데이터 전체 오류 (rmse) = 2.8430

시험 데이터 전체 오류 (R2-score) = 0.8765
```

XGBoost (Extreme Gradient Boosting) - (for classification) : 학습 과정

- AdaBoost는 여러 개의 약한 분류기를 사용하고, 잘못 분류한 데이터 샘플을 더 많이 샘플링해서 정확도를 높이려는 알고리즘이다.
- 처음에는 모든 데이터를 대상으로 1/N 비율로 동등하게 샘플링해서 서브-데이터를 만든 후 특정 분류기 (ex : DT, SVM 등)를 학습한다.

feature (X) target (Y)

Drug dosage	Drug effect	Residual (0)
3	No=0	-0.5
8	Yes=1	0.5
12	Yes=1	0.5
17	No=0	-0.5

1) 초기 추정치를 임의로 설정하고 (ex : 0.5)
residual (0)을 계산한다.

residual (0) = 0 - 0.5 = -0.5
1 - 0.5 = 0.5
1 - 0.5 = 0.5
0 - 0.5 = -0.5

2) residual (0)에 대해 similarity와 output value를 계산한다.

-0.5, 0.5, 0.5, -0.5 $\lambda = 1$을 적용했음.

$$similarity = \frac{(-0.5 + 0.5 + 0.5 - 0.5)^2}{0.5 * (1 - 0.5) + \cdots + 1} = 0$$

$$output = \frac{-0.5 + 0.5 + 0.5 - 0.5}{0.5 * (1 - 0.5) + \cdots + 1} = 0$$

$$L = \sum_{i=1}^{n} \underbrace{L(y_i, p_i)}_{\text{mse}} + \gamma T + \frac{1}{2}\lambda O^2$$

- T : 터미널 노드 개수
- O : output value
- λ : regularization constant
- γ : pruning 판단 상수

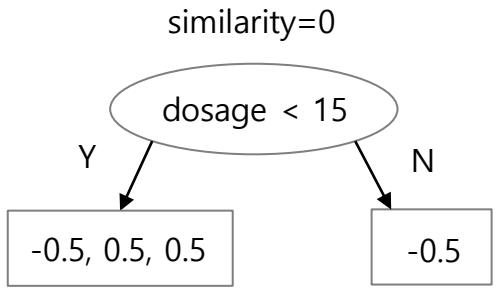
minimize w.r.t output value (γ는 생략했음)

$$output\ value(O) = \frac{\sum_{i=1}^{n} resudual}{\sum_{i=1}^{n} p_i(1 - p_i) + \lambda}$$

$$similarity\ score = \frac{(\sum_{i=1}^{n} resudual_i)^2}{\sum_{i=1}^{n} p_i(1 - p_i) + \lambda}$$

Loss가 최소가 되는 Tree의 조건

3) residual (0)에 대해 tree를 생성한다.



similarity=0.14 similarity=0.2

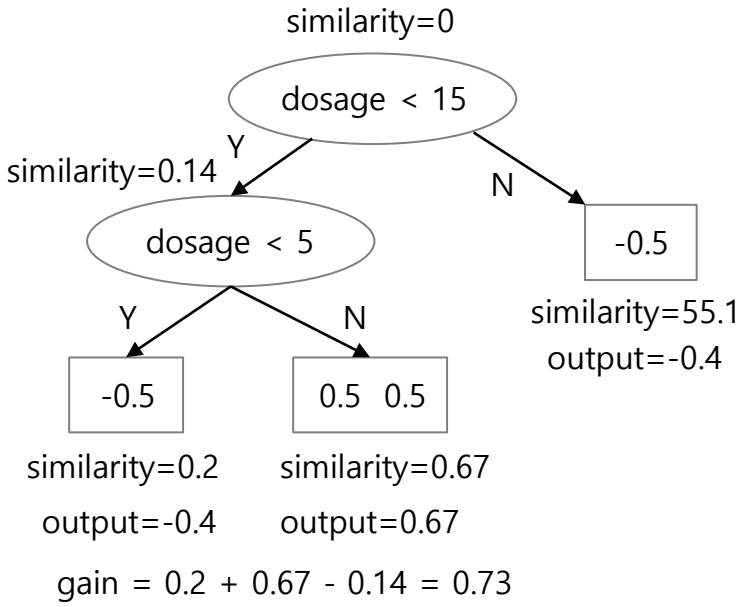
gain = 0.14 + 0.2 - 0 = 0.34

분할로 인해 0.34 만큼 이득이 생겼음.

- 각 leaf 노드의 similarity score를 계산한다.
- Gain이 가장 큰 분기를 찾아 tree를 생성한다. (weighted quantile search 방법)
- 이 경우는 dosage < 15 조건일 때의 gain이 가장 크다. Loss도 작아진다.
- leaf 노드에 residual이 유사한 것끼리 모여 있으면 similarity score가 높다. 상쇄되는 부분이 작기 때문이다.

XGBoost (Extreme Gradient Boosting) - (for classification) : 학습 과정

4) tree를 완성하고 leaf 노드의 output value를 계산한다.



gain = 0.2 + 0.67 - 0.14 = 0.73

- (-0.5, 0.5, 0.5) 노드를 gain이 가장 큰 조건으로 분기한다. (weighted quantile search)
- 이 경우는 dosage < 5 조건일 때의 gain이 가장 크다.
- (0.5 0.5) 노드는 분기해도 더 이상 이득이 없다 (gain < 0)
- Tree를 분기한 후 gain < γ라면 분기하지 않는다 (pruning)
- 예를 들어 γ = 0.5라면 좌측 서브 트리는 0.34 < 0.5이므로 (-0.5, 0.5, 0.5) 노드는 분리하지 않는다.
- γ는 분석자가 지정할 하이퍼 파라메터이다.

5) 좌측 tree의 output value를 이용하여 residual (1)을 계산한다.

- 초기 추정치 확률 = 0.5 → log(odds)를 계산한다.

$$\log\left(\frac{p}{1-p}\right) = log\left(\frac{0.5}{1-0.5}\right) = 0$$

학습률 (ε) = 0.3

- 첫 번째 데이터의 log(odds) 추정치 = 0 + 0.3 * (-0.4) = -0.12
- 첫 번째 데이터의 확률 추정치 = $\frac{1}{1+\exp(0.12)} = 0.47$ ← 초기 추정치인 0.5보다 작아졌다.
- new residual = 0 - 0.47 = -0.47

Drug dosage	Drug effect	Init. prob	Residual (0)	2nd prob	Residual (1)
3	No=0	0.5	-0.5	0.47	-0.47
8	Yes=1	0.5	0.5	0.55	0.45
12	Yes=1	0.5	0.5	0.55	0.45
17	No=0	0.5	-0.5	0.47	-0.47

← 추정치가 점점 정확해 지고, residual이 점점 작아지고 있다.

6) residual (1)으로 또 tree를 생성하고, residual이 줄어들지 않을 때까지 반복한다.

- 첫 번째 데이터의 log(odds) 추정치 = 초기 추정치 + ε * 이전 tree의 output + ε * 현재 tree의 output + ...

XGBoost : Classification 연습

* 실습 파일 : 3-8.XGBoost(classification).py

- XGBoost (classifier) 로 Breast Cancer 데이터를 학습하고 성능을 평가한다.

```
# XGBoost (classifier)로 Breast Cancer 데이터를 학습한다.
# GBM for classification
# xgboost 설치 : conda install -c anaconda py-xgboost
# ------------------------------------------------------
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier

# breast cancer 데이터를 가져온다.
cancer = load_breast_cancer()

# Train 데이터 세트와 Test 데이터 세트를 구성한다
trainX, testX, trainY, testY = \
    train_test_split(cancer['data'], cancer['target'], test_size = 0.2)

# XGBoost (classifier)로 Train 데이터 세트를 학습한다.
model = XGBClassifier(objective='binary:logistic')
model.fit(trainX, trainY)

# Test 세트의 Feature에 대한 class를 추정하고, 정확도를 계산한다
# accuracy = model.score(testX, testY)와 동일함.
predY = model.predict(testX)
accuracy = (testY == predY).mean()
print()
print("* 시험용 데이터로 측정한 정확도 = %.2f" % accuracy)

# Train 세트의 Feature에 대한 class를 추정하고, 정확도를 계산한다
predY = model.predict(trainX)
accuracy = (trainY == predY).mean()
print("* 학습용 데이터로 측정한 정확도 = %.2f" % accuracy)
```

결과 :

```
* 시험용 데이터로 측정한 정확도 = 0.97
* 학습용 데이터로 측정한 정확도 = 1.00
```

XGBoost : Classification 연습

* 실습 파일 : 3-9.XGBoost(classification-2).py

- XGBoost (classifier) 로 iris 데이터를 학습하고 성능을 평가한다. 학습데이터와 시험데이터를 xgb 데이터 형태로 변환한 후 학습한다.

```
# XGBoost (classifier)로 iris 데이터를 학습한다.
# 학습데이터와 시험데이터를 xgb 데이터 형태로 변환 후 학습.
# ------------------------------------------------------
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import xgboost as xgb
import numpy as np

# iris 데이터를 가져온다.
iris = load_iris()

# Train 데이터 세트와 Test 데이터 세트를 구성한다
trainX, testX, trainY, testY = \
    train_test_split(iris['data'], iris['target'], test_size = 0.2)

# XGBoost (classifier)로 Train 데이터를 학습한다.
# 학습데이터와 시험데이터를 xgb의 데이터 형태로 변환한다.
trainD = xgb.DMatrix(trainX, label = trainY)
testD = xgb.DMatrix(testX, label = testY)

param = {
    'eta': 0.3,
    'max_depth': 3,
    'objective': 'multi:softprob',  # softmax
    'num_class': 3}   # class 개수 = 3개 (multi class classification)

model = xgb.train(params = param, dtrain = trainD, num_boost_round = 20)

# Test 세트의 Feature에 대한 class를 추정하고, 정확도를 계산한다
predY = model.predict(testD)
predY = np.argmax(predY, axis=1)
accuracy = (testY == predY).mean()
print()
print("* 시험용 데이터로 측정한 정확도 = %.2f" % accuracy)

# Train 세트의 Feature에 대한 class를 추정하고, 정확도를 계산한다
predY = model.predict(trainD)
predY = np.argmax(predY, axis=1)
accuracy = (trainY == predY).mean()
print("* 학습용 데이터로 측정한 정확도 = %.2f" % accuracy)
```

결과 :

```
* 시험용 데이터로 측정한 정확도 = 0.97
* 학습용 데이터로 측정한 정확도 = 1.00
```

Light GBM

- AdaBoost는 가중치를 이용하여 데이터를 샘플링 하지만, GBM, XGBoost는 데이터에 가중치를 부여하는 알고리즘이 없으므로 샘플링 과정이 없고, 전체 데이터의 residual을 대상으로 트리를 만든다.
- GBM/XGBoost는 전체 데이터를 대상으로 트리를 만들기 때문에 information gain이 큰 분기 (split)를 찾는데 시간이 오래 걸린다.
- Light GBM은 residual의 크기를 이용해서 (가중치 역할) 데이터를 샘플링할 수 있는 알고리즘이다. 게다가 feature들을 병합해서 (일종의 feature reduction) feature의 개수를 줄인다. 이렇게 하면 residual tree를 만드는데 시간을 대폭 절약할 수 있다.
- Light GBM은 2017년 Guolin Ke 등이 "LightGBM: A Highly Efficient Gradient Boosting Decision Tree" 이라는 논문으로 발표한 것이다. 속도가 빠르면서도 XGBoost에 비해 성능이 떨어지지 않아 최근 매우 관심을 받고 있는 알고리즘이다.
- 핵심 기능으로는 데이터 샘플링을 위한 GOSS (Gradient-based One-Side Sampling)와 feature 개수를 줄이기 위한 EFB (Exclusive Feature Bundling)가 있다.

LightGBM: A Highly Efficient Gradient Boosting Decision Tree

Guolin Ke1, Qi Meng2, Thomas Finley3, Taifeng Wang1, Wei Chen1, Weidong Ma1, Qiwei Ye1, Tie-Yan Liu1

Abstract

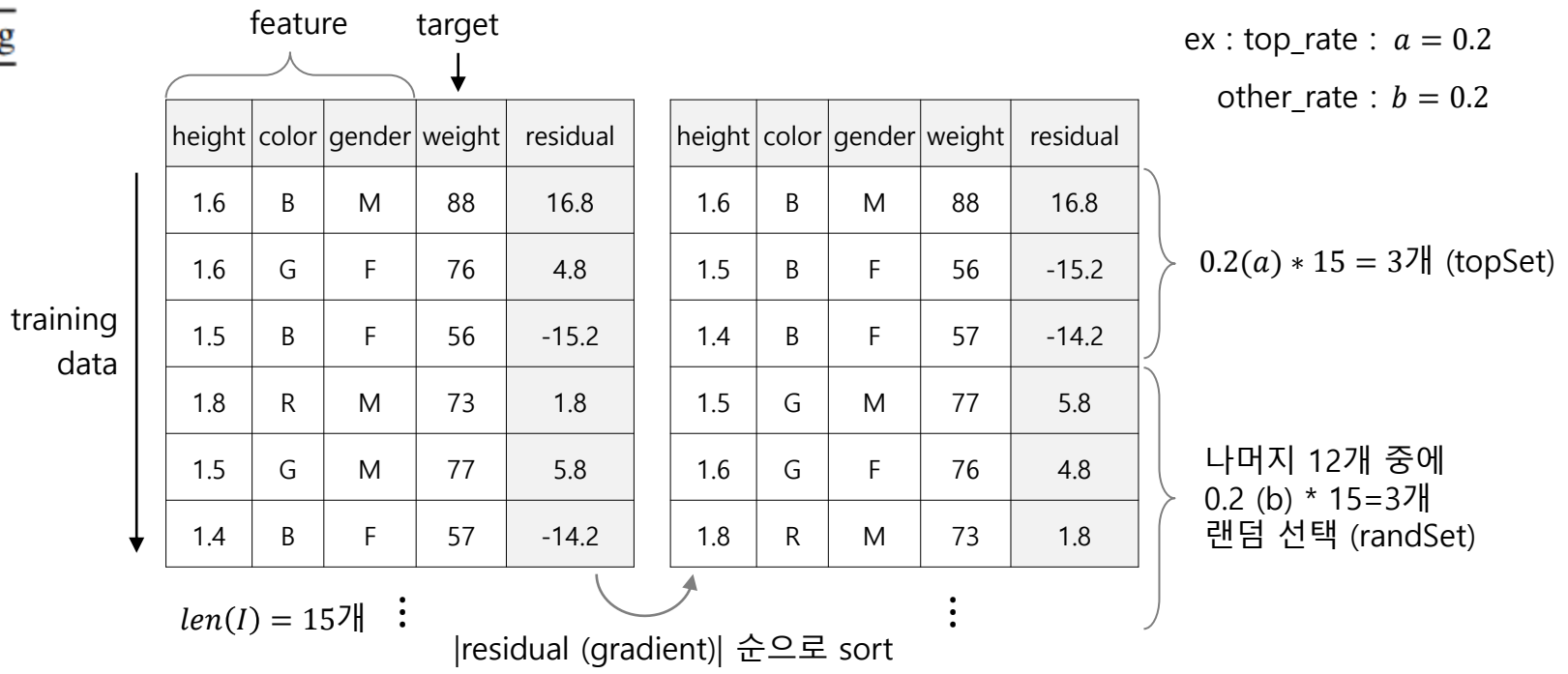
Gradient Boosting Decision Tree (GBDT) is a popular machine learning algorithm, and has quite a few effective implementations such as XGBoost and pGBRT. Although many engineering optimizations have been adopted in these implementations, the efficiency and scalability are still unsatisfactory when the feature dimension is high and data size is large. A major reason is that for each feature, they need to scan all the data instances to estimate the information gain of all possible split points, which is very time consuming. To tackle this problem, we propose two novel techniques: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). With GOSS, we exclude a significant proportion of data instances with small gradients, and only use the rest to estimate the information gain. We prove that, since the data instances with larger gradients play a more important role in the computation of information gain, GOSS can obtain quite accurate estimation of the information gain with a much smaller data size. With EFB, we bundle mutually exclusive features (i.e., they rarely take nonzero values simultaneously), to reduce the number of features. We prove that finding the optimal bundling of exclusive features is NP-hard, but a greedy algorithm can achieve quite good approximation ratio (and thus can effectively reduce the number of features without hurting the accuracy of split point determination by much). We call our new GBDT implementation with GOSS and EFB LightGBM. Our experiments on multiple public datasets show that, LightGBM speeds up the training process of conventional GBDT by up to over 20 times while achieving

Light GBM : Gradient-based One-Side Sampling (GOSS)

- 트리에 적용할 데이터를 샘플링한다. 샘플링할 때 gradient (residual 혹은 loss) 값을 이용해서 중요한 데이터는 최대한 보존하고, 중요하지 않은 데이터는 샘플링 되지 않도록 (일종의 샘플링 가중치) 조절한다. Gradient가 클수록 학습이 더 필요하므로 학습 데이터에 포함시키고, gradient가 작은 데이터는 학습 데이터에서 제외하도록 샘플링한다.
- GOSS는 아래 알고리즘으로 a와 b (하이퍼) 파라메터를 이용한다. 아래 예시에서 학습 데이터는 topSet + randSet : 6개이다. 전체 15개 데이터로 트리를 구축하지 않고, 6개 데이터만으로 트리를 구축하므로 속도가 빠르다. 이렇게 해도 성능은 거의 저하되지 않는 것으로 알려져 있다.

Algorithm 2: Gradient-based One-Side Sampling

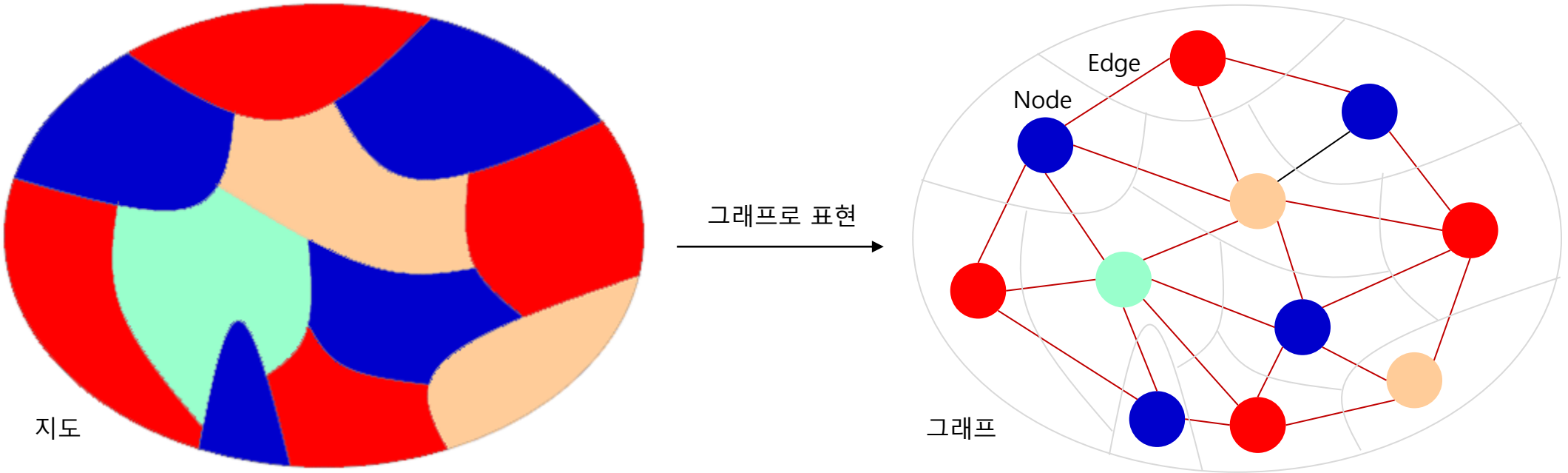
Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models ← { }, fact ← $\frac{1-a}{b}$
topN ← a × len(I) , randN ← b × len(I)
for $i = 1$ ***to*** d **do**
  preds ← models.predict(I)
  g ← $loss$(I, preds), w ← {1,1,...}
  sorted ← GetSortedIndices(abs(g))
  topSet ← sorted[1:topN]
  randSet ← RandomPick(sorted[topN:len(I)], randN)
  usedSet ← topSet + randSet
  w[randSet] × = fact ▷ Assign weight $fact$ to the small gradient data.
  newModel ← L(I[usedSet], − g[usedSet], w[usedSet])
  models.append(newModel)



$$fact = \frac{1-a}{b} = \frac{1-0.2}{0.2} = 4 \ (>1)$$ ← randSet에는 information gain을 계산할 때 4를 곱해준다. (12개 중 1/4인 3개만 사용했기 때문)

이산수학 : Graph coloring problem (그래프 채색 문제)

- 아래 왼쪽 지도의 각 영역을 구별할 수 있게 색칠하려면 최소 몇 가지 색이 필요할까?
- 지도를 오른쪽 그림처럼 그래프 (node와 edge)로 표현하고, 연결된 노드들이 다른 색을 갖게 색칠하려면 최소 4개의 색이 필요하다.
- 임의의 그래프에 대해 최소 색상수를 찾는 문제를 graph coloring problem이라 한다. (쉽지 않다)
- Light GBM에서는 feature 개수를 줄이기 위해 graph coloring problem을 적용했다. Feature들을 노드로 보고, 두 feature의 값이 동시에 0이 아닌 (conflict) 개수를 엣지 (edge)로 사용한다. 그러면 최소 색상수가 줄어든 feature의 개수가 된다.
- 아래 경우는 총 11개의 영역이 있고, 4개의 색상이 필요했다. 각 영역을 학습 데이터의 feature 개수 11개라면, 4개의 feature로 줄일 수 있다.
- Light GBM의 EFB (Exclusive Feature Bundling)는 이 원리를 이용해서 학습 데이터의 feature 개수를 줄인다. 학습 속도가 빨라진다.



예시 출처 : 고려대학교 강성필 교수 영상 (https://www.youtube.com/watch?v=4C8SUZJPlMY)

Light GBM : Exclusive Feature Bundling (EFB) - (1) Greedy Bundling

- EFB는 feature의 개수를 줄이는 알고리즘으로 graph coloring problem을 적용한다.
- Feature 개수가 많은 학습 데이터에는 '0'이 많이 포함된 경우가 많다. 극단적인 예시로는 원-핫 인코딩이 있다. 원-핫 인코딩은 1개 feature만 값이 '1'이고, 나머지는 모두 '0'이다. 이런 형태의 학습 데이터를 희소행렬 (sparse matrix)이라 한다.
- 희소행렬의 경우 두 feature의 값이 모두 '0'이 아닌 경우 (conflict라 한다)가 많지 않다면, 이 feature들을 하나로 병합 (bundling)할 수 있다. ← Algorithm 3
- 두 feature의 값 중 1개라도 '0'이 있으면 두 feature는 상호 베타적 (exclusive feature)이라 하고, 이들을 합쳐서 feature 개수를 줄이는 것이다.
- 아래 예시에서 F1과 F2 (F : feature)의 conflict count는 6개이다. Conflict count를 갖는 feature 행렬을 만든 후 conflict count의 합 (degree라 한다)이 큰 순서로 정렬한다 (searchOrder). 그리고 searchOrder 행렬의 feature를 순차적으로 검색해서 그래프를 만든다. F5 → F1 → F2 → F3 → F4 순.

Algorithm 3: Greedy Bundling

Input: F: features, K: max conflict count
Construct graph G
searchOrder ← G.sortByDegree()
bundles ← { }, bundlesConflict ← { }
for i ***in*** *searchOrder* **do**
  needNew ← True
  for $j = 1$ ***to*** *len(bundles)* **do**
    cnt ← ConflictCnt(bundles[j],F[i])
    if *cnt + bundlesConflict[i]* $\leq K$ **then**
      bundles[j].add(F[i]), needNew ← False
      break
  if *needNew* **then**
    Add $F[i]$ as a new bundle to *bundles*
Output: *bundles*

	F1	F2	F3	F4	F5
i1	1	1	0	0	1
i2	0	0	1	1	1
i3	1	2	0	0	2
i4	0	0	2	3	1
i5	2	1	0	0	3
i6	3	3	0	0	1
i7	0	0	3	0	2
i8	1	2	3	4	3
i9	1	0	1	0	0
i10	2	3	0	0	2

F1과 F2의 conflict count = 6개

conflict count 행렬

	F1	F2	F3	F4	F5
F1		6	2	1	6
F2	6		1	1	6
F3	2	1		3	4
F4	1	1	3		3
F5	6	6	4	3	
sum	15	14	10	8	19

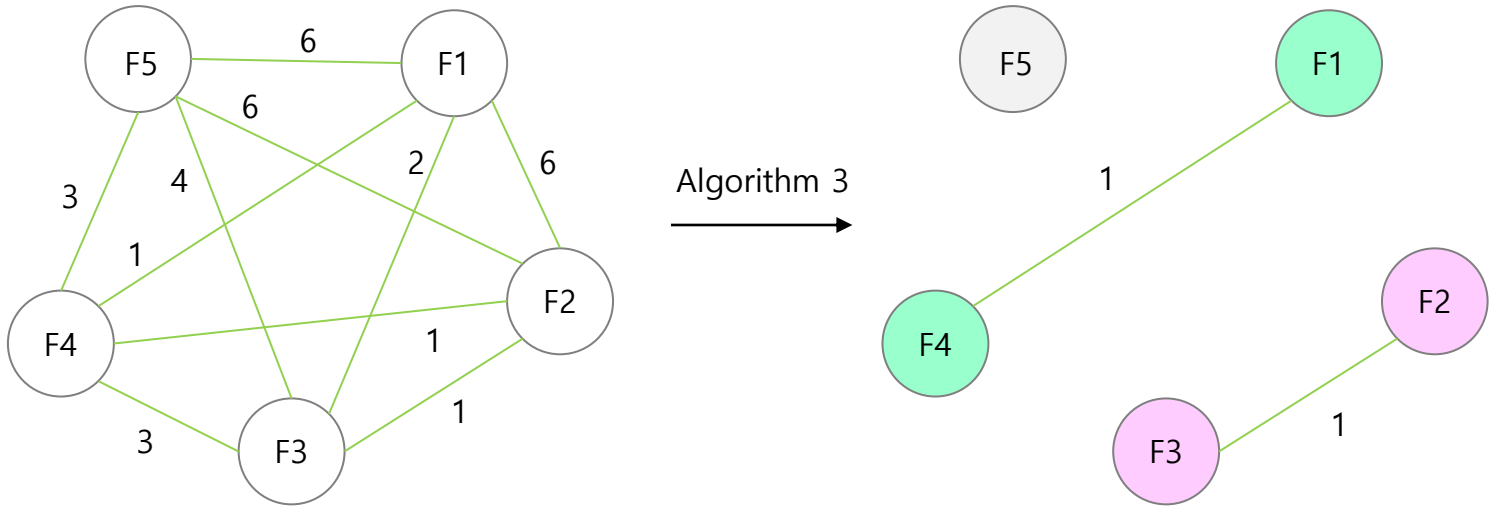
sum이 큰 순서로 정렬 (d : degree)

searchOrder

	F5	F1	F2	F3	F4
d	19	15	14	10	8

Light GBM : Exclusive Feature Bundling (EFB) - (1) Greedy Bundling

- 이전 페이지의 conflict count 행렬을 이용해서 아래와 같이 그래프를 그린다. 노드는 feature이고 엣지는 conflict count이다.
- Max conflict count (K)를 설정하고 (ex : 2개) searchOrder 순서대로 그래프를 탐색하면서 Algorithm-3 (Greedy Bundling)에 따라 graph coloring을 수행한다.
- 아래 예시에서는 conflict count가 2 이하인 엣지만 남기고 모든 연결을 제거했다.
- 아래 예시에서 F2와 F4는 conflict count = 1로 연결돼 있지만 F4는 이미 F1과 conflict count = 1로 연결돼 있으므로 F2와 F4의 연결은 제거한다.
- Graph coloring에 의해 3가지 색이 결정되었으며, 연결된 feature들을 하나로 묶어 {F5}, {F1, F4}, {F2, F3} 3개의 그룹을 만든다. 그러면 최초 5개 feature를 3개로 줄일 수 있다.



	F1	F2	F3	F4	F5
i1	1	1	0	0	1
i2	0	0	1	1	1
i3	1	2	0	0	2
i4	0	0	2	3	1
i5	2	1	0	0	3
i6	3	3	0	0	1
i7	0	0	3	0	2
i8	1	2	3	4	3
i9	1	0	1	0	0
i10	2	3	0	0	2

K : max conflict count = N * cut-off = 데이터 10개 * 0.2 = 2

{F2, F3}를 하나로 묶고, {F1, F4}를 하나로 묶을 수 있다

예시 출처 : 고려대학교 강성필 교수 영상 (https://www.youtube.com/watch?v=4C8SUZJPlMY)

Light GBM : Exclusive Feature Bundling (EFB) - (2) Merge Exclusive Features

- Greedy Bundling으로 묶은 feature들을 병합 (merge)한다. ← Algorithm 4
- 아래 feature 값들은 원래 실숫값으로 구성된 학습 데이터를 histogram-based algorithm (논문의 Algorithm 1) 으로 만든 bin number이다. F1은 bin 개수가 4개이고 number가 0 부터 3이다 [0, 3]. (F1.numBin = 3) 이 경우는 한 번들의 feature가 2개이므로 binRanges = 3이 된다. (첫 번째 for 문은 F-1 까지 돈다)
- {F1, F4}을 묶어서 하나의 feature로 만든다 (F1,4). F1을 기준으로 F1의 bin값 (F1.bin)이 0이 아니면 F1,4.bin = F1.bin으로 적용하고, F1.bin이 0이면 F1,4.bin = F4.bin + binRange을 적용한다. binRange는 일종의 offset 값이다. 즉 F1이 0인 값은 F1의 마지막 bin 값 이후 F4.bin 번째 값을 갖도록 하는 것이다.
- 아래 과정을 거치면 5개 feature가 3개로 줄어든다.

Algorithm 4: Merge Exclusive Features

Input: $numData$: number of data
Input: F: One bundle of exclusive features
binRanges ← {0}, totalBin ← 0
for f ***in*** F **do**
    totalBin += f.numBin
    binRanges.append(totalBin)
newBin ← new Bin(numData)
for $i = 1$ ***to*** $numData$ **do**
    newBin[i] ← 0
    for $j = 1$ ***to*** $len(F)$ **do**
        if $F[j].bin[i] \neq 0$ **then**
            newBin[i] ← F[j].bin[i] + binRanges[j]
Output: $newBin, binRanges$

	F1	F2	F3	F4	F5
i1	1	1	0	0	1
i2	0	0	1	1	1
i3	1	2	0	0	2
i4	0	0	2	3	1
i5	2	1	0	0	3
i6	3	3	0	0	1
i7	0	0	3	0	2
i8	1	2	3	4	3
i9	1	0	1	0	0
i10	2	3	0	0	2

merge →

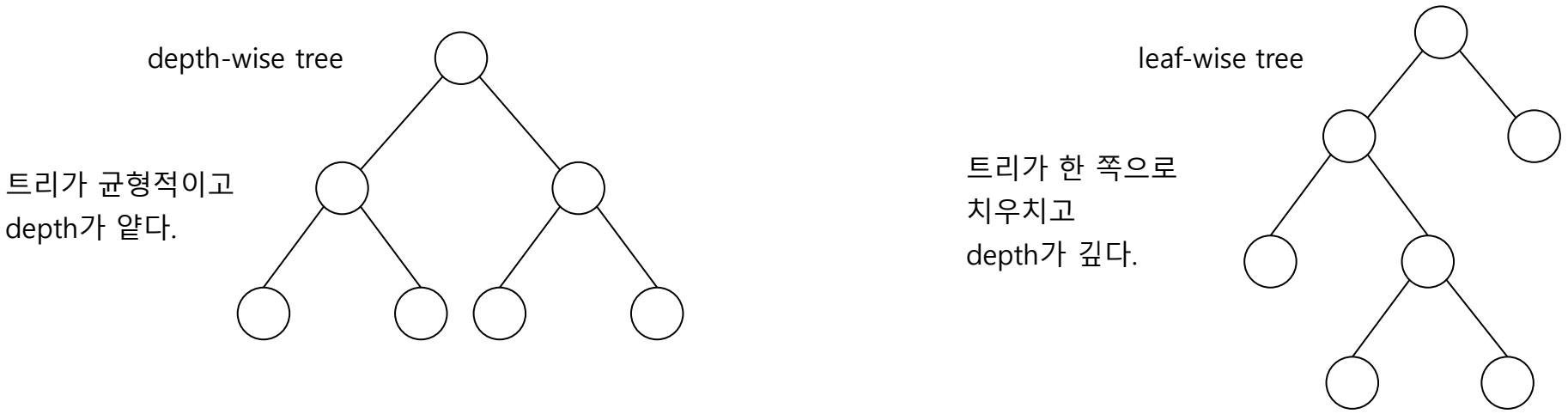
	F1,4	F2,3	F5
i1	1	1	1
i2	4	4	1
i3	1	2	2
i4	6	5	1
i5	2	1	3
i6	3	3	1
i7	0	6	2
i8	1	2	3
i9	1	4	0
i10	2	3	2

F2.bin = 0이므로,
F3.bin + offset = 1 + 3

conflict인 경우는
F3.bin 값인 3을 버리고
F2.bin 값을 사용한다.
이런 경우는 많지 않다고
간주한다. (감수해야 하는
부분임)

Depth-wise vs. leaf-wise tree

- 일반적으로 Decision Tree 알고리즘은 depth-wise (level-wise 라고도 함)형태로 트리를 만든다. 트리가 균형적이며 depth가 깊지 않다. 이렇게 하는 이유는 오버피팅을 줄이고 일반화 특성을 좋게 하기 위함이다 (regularization). 그러나 데이터가 많으면 트리를 만드는데 시간이 오래 걸린다.
- Leaf-wise (best-first 방식이라고도 함)는 트리를 만들 때 전체적인 균형보다는 loss가 작아지는 쪽으로 분기하는 방식이다. 데이터가 많은 경우 속도가 빠르지만, 데이터가 적은 경우 오버피팅에 취약하다.
- Leaf-wise는 2007년 Haijan Shi가 "Best-first decision tree learning"라는 논문으로 제안한 방식이다.
- XGBoost는 기본적으로 depth-wise 방식을 사용한다. 따라서 데이터가 적은 경우에도 무리 없이 동작한다. XGBoost에서 leaf-wise 방식을 사용하려면 파라메터 tree_method = hist, grow_policy = lossguide를 사용한다.
- Light GBM은 기본적으로 leaf-wise 방식을 사용한다. Light GBM은 많은 데이터를 빠르게 처리하는 것이 목적이고 데이터가 많으면 오버피팅의 가능성이 줄어들기 때문에 leaf-wise 방식을 사용한다. 공식 문서에 의하면 데이터가 10,000개 이상인 경우를 "많다"라고 한다. 만약 데이터가 적은 경우에 Light GBM을 사용하면 오버피팅에 취약할 수 있다.



Light GBM : Classification 연습

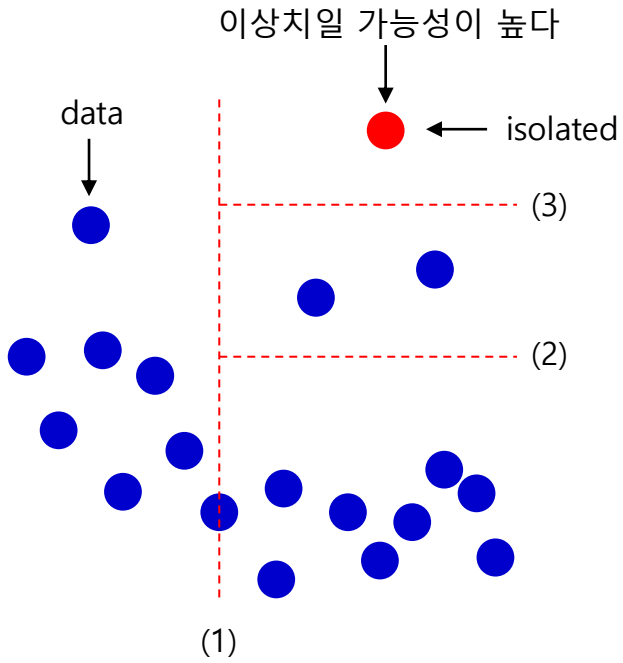
* 실습 파일 : 3-10.lightgbm(classification).py

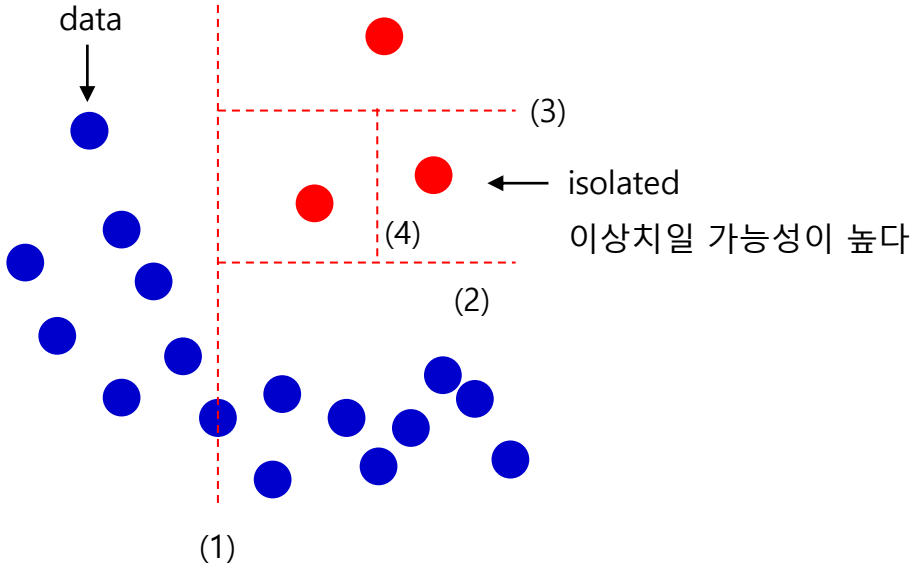
- Light GBM으로 Kaggle의 Santander Customer Satisfaction 데이터를 학습한다.

```
# Light GBM으로 Kaggle의 Santander Customer Satisfaction 데이터를 학습한다.
# ------------------------------------------------------------------------
import pandas as pd
from sklearn.model_selection import train_test_split
from lightgbm import LGBMClassifier
from sklearn.metrics import roc_auc_score

df = pd.read_csv("dataset/santander.csv", encoding='latin-1')
df.info()
df.describe()
df['TARGET'].value_counts()
df['var3'].value_counts()

# 'var3' feature의 -999999를 2로 치환하고, 'ID' feature는 drop한다.
df['var3'].replace(-999999, 2, inplace=True)
df.drop('ID', axis = 1, inplace=True)

# 피처와 레이블 세트를 분리한다.
X_features = df.iloc[:, :-1]
y_labels = df.iloc[:, -1]

# 학습, 평가, 시험 데이터를 구성한다.
X_train, X_test, y_train, y_test = train_test_split(X_features, y_labels,
                                                    test_size = 0.2)

X_train, X_eval, y_train, y_eval = train_test_split(X_train, y_train,
                                                    test_size = 0.2)

# 레이블 분포가 고른지 확인한다.
y_train.value_counts() / len(y_train)
y_test.value_counts() / len(y_test)
y_eval.value_counts() / len(y_eval)

# 모델 생성 (use_label_encoder는 future warining을 없애기 위함)
lgb = LGBMClassifier(n_estimators = 100, boosting="goss", top_rate=0.2,
                     other_rate=0.1)

# 학습
lgb.fit(X_train, y_train, early_stopping_rounds=100,
        eval_set=[(X_train, y_train), (X_eval, y_eval)], eval_metric="auc")

# 평가. 레이블이 고르지 않으므로 ROC AUC로 평가한다.
pred = lgb.predict_proba(X_test)[:, 1]
auc = roc_auc_score(y_test, pred)
print("ROC AUC = {0:.4f}".format(auc))

[332] training's auc: 0.96376 training's binary_logloss: 0.0811683
                 valid_1's auc: 0.791109 valid_1's binary_logloss: 0.154168
Early stopping, best iteration is:
[32] training's auc: 0.889834 training's binary_logloss: 0.121413
                valid_1's auc: 0.822207 valid_1's binary_logloss: 0.140356

ROC AUC = 0.8336
```

Isolation Forest (iForest) : 학습 과정의 직관적 이해

- iForest는 2008년 Fei Tony Liu, Kai Ming Ting and Zhi-Hua Zhou가 제안한 unsupervised learning 형태의 알고리즘이다.
- iForest는 이상 데이터 검출 (anomaly detection, outlier detection, fraud detection) 등에 활용될 수 있다.
- 기본 아이디어는 매우 단순하다. Decision tree로 데이터를 학습하면 이상 데이터 (outlier)는 빨리 분기되어 leaf node에 홀로 남게 될 가능성이 높아진다.
- 정상 데이터는 일정 범위안에 모여 있으므로 leaf node에 홀로 남으려면 tree의 depth가 깊어진다.
- 아래 왼쪽 그림의 경우 분기 (3)에 의해 이상 데이터가 leaf node에 홀로 남게 된다 (isolate 됨). 이 노드의 depth는 2이다. 다른 데이터들은 더 많은 분기가 필요하므로 depth가 깊어진다. 오른쪽 그림의 경우 (4)까지 분기하면 두 개 데이터가 isolate되고, 이 데이터들도 이상치일 가능성이 있다.
- 실제는 Random Forest를 구성하고 isolate된 leaf node 마다 anomaly score를 계산한 후 평균을 계산해서 score → 1.0에 가까운 데이터를 이상 데이터로 판정한다.

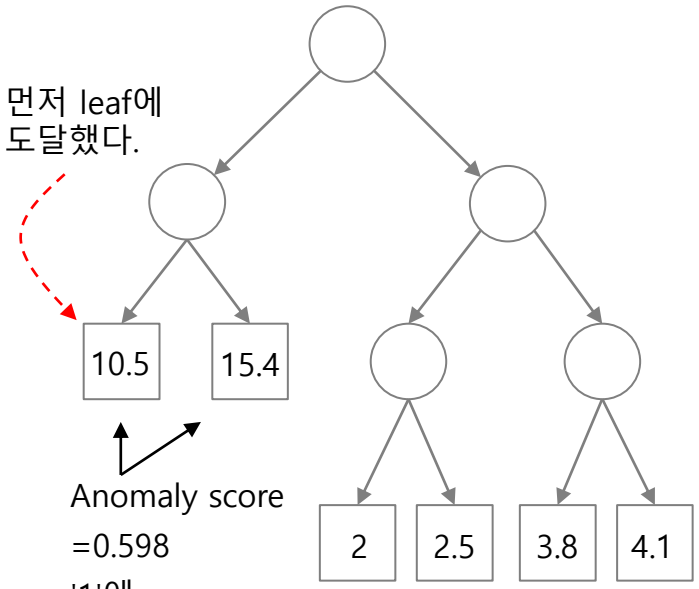




Isolation Forest (iForest) : Anomaly score

- Anomaly score는 이진검색트리 (Binary Search Tree : BST)에서 사용되는 c(n)의 개념을 차용해서 아래와 같이 계산한다.
- Anomaly score가 '1'에 가까울수록 이상데이터일 가능성이 높고, '0'에 가까울수록 정상 데이터일 가능성이 높다. '0.5'에 가까우면 정상데이터와 이상데이터의 구분이 명확하지 않다는 의미다. 아래 데이터의 경우는 10.5와 15.4가 이상 데이터일 가능성이 높다.
- 아래 tree에서 이상 데이터와 정상 데이터의 depth가 1개 밖에 차이가 나지않으므로 score = 0.46으로 0.5에 가깝게 나왔다.

데이터 (X) = [2, 2.5, 3.8, 4.1, 10.5, 15.4]



◯ : internal node (n-1 = 5개)

□ : external node (n = 6개)

1) average path length of unsuccessful search in BST
- leaf node까지의 평균 depth
- 이진검색트리(BST)에서 leaf node까지 탐색해도 데이터를 찾지 못함

$$c(n) = 2H(n-1) - \left(\frac{2(n-1)}{n}\right)$$

$$H(i) = \log(i) + 0.5772156649\ (Euler's\ constant)$$

$$\text{평균 depth} = \frac{2+2+3+3+3+3}{6} = 2.67$$

$$c(n) = 2 * (\log(5) + 0.577) - \frac{2*5}{6} = 2.70$$

비슷함

2) Anomaly score

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}}$$

h(x)는 데이터 x의 path length
E(h(x))는 여러 tree의 h(x) 평균

$$s(2, n=6) = 2^{-\frac{3}{2.7}} = 0.46$$

$$s(10.5, n=6) = 2^{-\frac{2}{2.7}} = 0.598$$

$$s(15.4, n=6) = 2^{-\frac{2}{2.7}} = 0.598$$

$$E(h(x)) \to c(n), s \to 0.5$$

$$E(h(x)) \to 0\ s \to 1$$

$$E(h(x)) \to n-1, s \to 0$$

Isolation Forest (iForest) : Anomaly score

* 실습 파일 : 3-11.IsolationForest(score).py

- sklearn의 IsolationForest를 사용해서 간단한 데이터에 대한 anomaly score를 확인한다. 이전 페이지의 결과와 비교해 본다.

```
# sklearn의 IsolationForest를 사용해서 anomaly score를 확인한다.
# ------------------------------------------------------------
from sklearn.ensemble import IsolationForest
import numpy as np

# 데이터. shape = (-1, 1) ~ 6행 1열로 맞춘다.
X = np.array([2, 2.5, 3.8, 4.1, 10.5, 15.4]).reshape(-1, 1)

# iForest 모델을 생성한다. score 확인을 위해 tree 개수는 1개로 한다.
# tree가 1개이기 때문에 실행할 때마다 결과가 불안정할 수 있다.
model = IsolationForest(n_estimators=1)

# iForest 모델을 이용하여 데이터 (X)를 학습한다.
model.fit(X)

# 판정 결과를 확인한다. 1 : 정상, -1 : 이상 데이터
pred = model.predict(X)
print("\n판정 결과 : tree = 1개")
print(pred)

# anomaly score를 확인한다.
score = abs(model.score_samples(X))
print("\nAnomaly score :")
print(np.round(score, 3))

# 50개 tree를 사용해서 다시 계산해 본다. 결과가 안정적이다.
model = IsolationForest(n_estimators=50)

# iForest 모델을 이용하여 데이터 (X)를 학습한다.
model.fit(X)

# 판정 결과를 확인한다. 1 : 정상, -1 : 이상 데이터
pred = model.predict(X)
print("\n판정 결과 : tree = 50개")
print(pred)

# anomaly score를 확인한다.
score = abs(model.score_samples(X))
print("\nAnomaly score :")
print(np.round(score, 3))
```

```
판정 결과 : tree = 1개 ← tree = 1개로 결과가 불안정할 수 있다.
[ 1  1  1  1 -1 -1]

Anomaly score :
[0.464 0.464 0.464 0.464 0.599 0.599] ← 이전 페이지 결과와 잘 일치한다.

판정 결과 : tree = 50개
[ 1  1  1  1 -1 -1]     ← tree = 결과가 안정적이다.

Anomaly score :
[0.476 0.42  0.401 0.418 0.558 0.634] ← 50개 tree의 평균적인 score
```

이전 페이지와 다른 tree가 생성되면 이 값이 다를 수 있다. (여러 번 실행으로 확인)

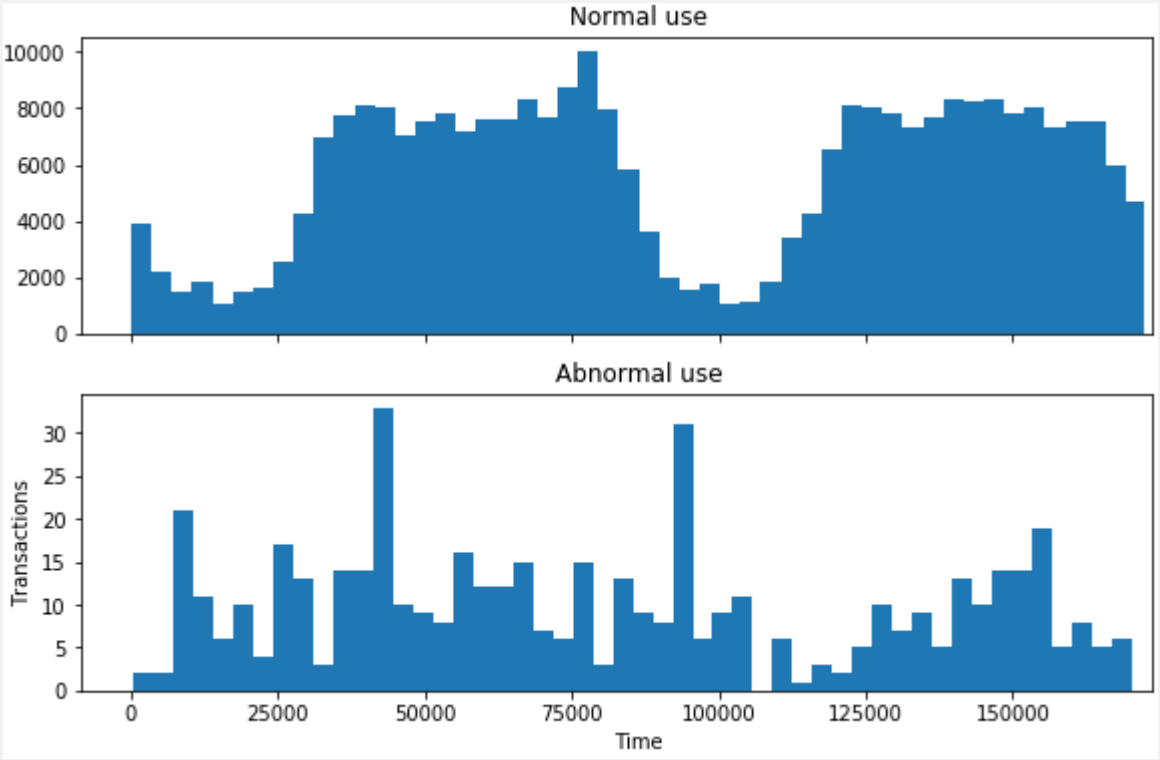
Isolation Forest (iForest) : Credit card fraud detection

* 실습 파일 : 3-12.IsolationForest(credit).py

- iForest를 사용한 credit card fraud detection 예시

```
# iForest를 사용한 credit card fraud detection 예시
# dataset : https://www.kaggle.com/mlg-ulb/creditcardfraud
# ---------------------------------------------------------
from sklearn.ensemble import IsolationForest
from sklearn.metrics import confusion_matrix, classification_report,
                            accuracy_score
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Credit data set을 읽어온다
# 사용 시간 (Time), 사용 금액 (Amount)을 정상 여부 (Class)를
# 제외한 나머지 feature들은 PCA로 변환된 수치임.
# Class (0 : 정상 사용, 1 : 비정상 사용 (fraud))
df = pd.read_csv('dataset/creditcard(fraud).csv')

# 사용 시간대별 트랜잭션의 분포를 확인해 본다.
f, (ax1, ax2) = plt.subplots(2,1, sharex=True, figsize=(10, 6))
ax1.hist(df['Time'][df['Class'] == 0], bins=50)
ax2.hist(df['Time'][df['Class'] == 1], bins=50)
plt.xlabel('Time')
plt.ylabel('Transactions')
ax1.set_title('Normal use')
ax2.set_title('Abnormal use')
plt.show()
```

```
df
Time         V1         V2  ...                 V28    Amount    Class
0           0.0  -1.359807  -0.072781  ... -0.021053  149.62      0
1           0.0   1.191857   0.266151  ...  0.014724    2.69      0
2           1.0  -1.358354  -1.340163  ... -0.059752  378.66      0
3           1.0  -0.966272  -0.185226  ...  0.061458  123.50      0
```

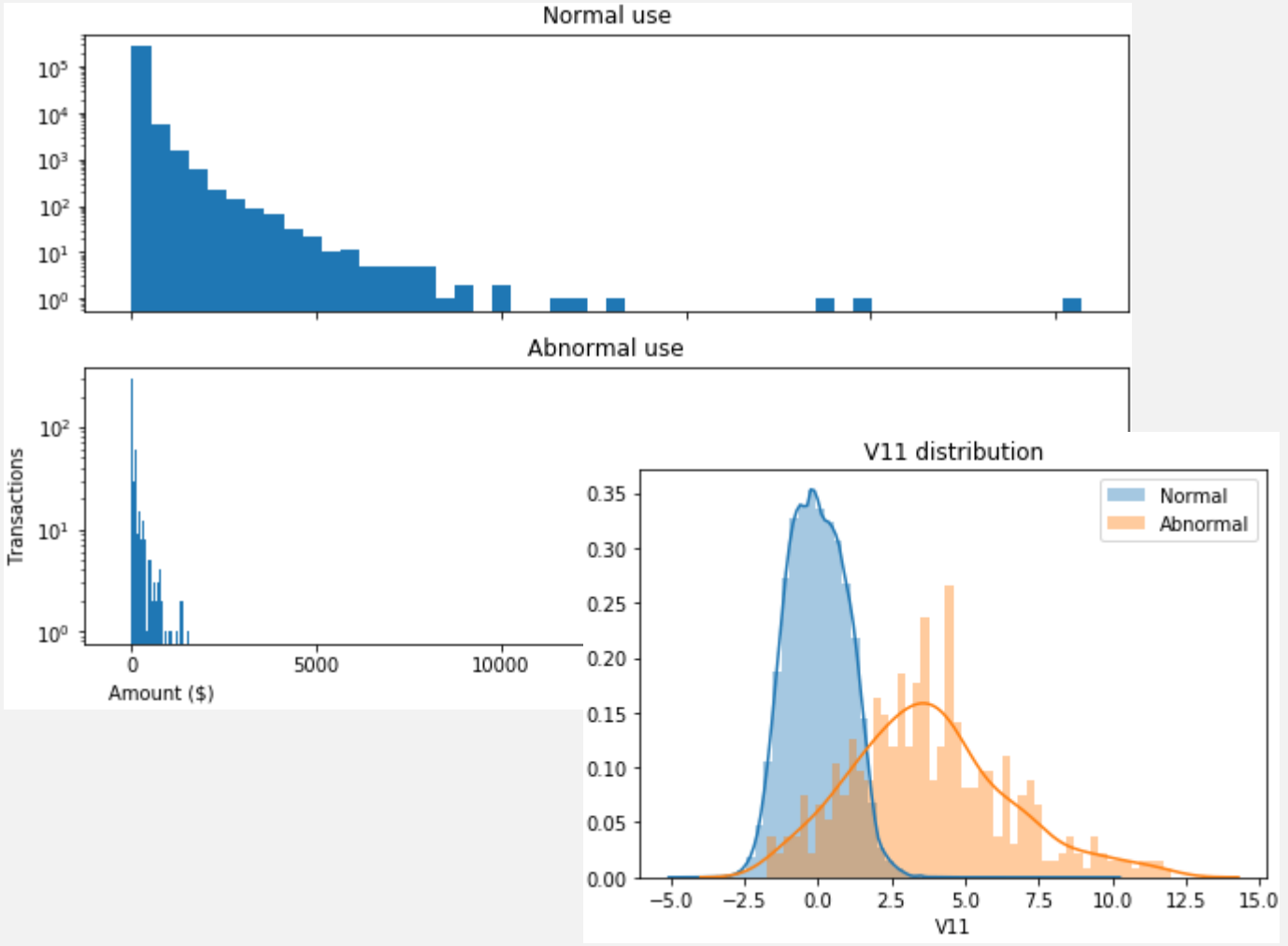


Isolation Forest (iForest) : Credit card fraud detection

* 실습 파일 : 3-12.IsolationForest(credit).py

- iForest를 사용한 credit card fraud detection 예시

```
# 사용 금액별 트랜잭션의 분포를 확인해 본다
f, (ax1, ax2) = plt.subplots(2,1, sharex=True, figsize=(10,6))
ax1.hist(df['Amount'][df['Class'] == 0], bins=50)
ax2.hist(df['Amount'][df['Class'] == 1], bins=50)
ax1.set_yscale('log')
ax2.set_yscale('log')
ax1.set_title('Normal use')
ax2.set_title('Abnormal use')
plt.xlabel('Amount ($)')
plt.ylabel('Transactions')
plt.show()

# feature별 분포를 확인해 본다.
feature = 'V11'
sns.distplot(df[feature][df['Class'] == 0], bins=50)
sns.distplot(df[feature][df['Class'] == 1], bins=50)
plt.legend(['Normal','Abnormal'], loc='best')
plt.title(feature + ' distribution')
plt.show()
```

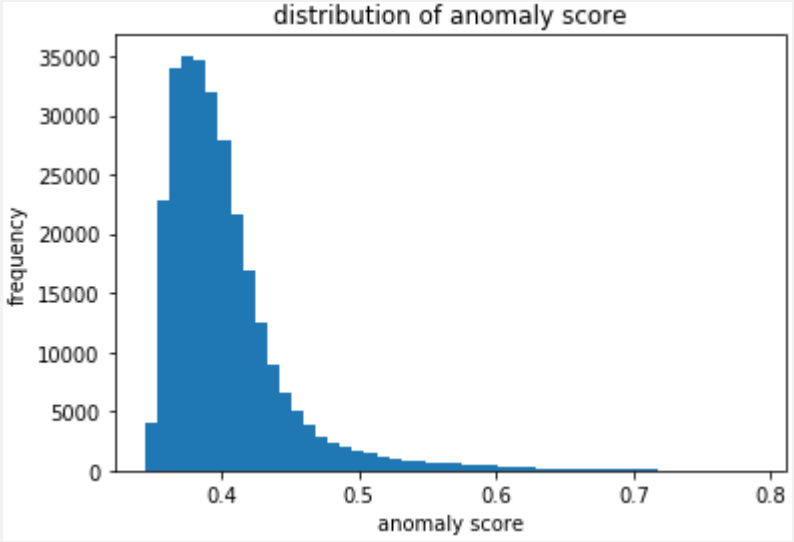


Isolation Forest (iForest) : Credit card fraud detection

* 실습 파일 : 3-12.IsolationForest(credit).py

- iForest를 사용한 credit card fraud detection 예시

```
# 학습 데이터를 만든다.
credit = np.array(df)
trainX = credit[:, :-1]
trainY = credit[:, -1]

# 비정상 사례의 비율을 확인해 본다.
normal = (trainY == 0).sum()
fraud = (trainY == 1).sum()
print("\n정상 사례 = ", normal)
print("비정상 사례 = ", fraud)
print("비정상 사례 비율 = %.4f (%%)" % (100 * fraud / normal))

# iForest로 이상치를 확인한다.
model = IsolationForest(n_estimators = 100)

# iForest 모델을 이용하여 데이터를 학습한다.
model.fit(trainX)

# Anomaly score를 확인한다.
score = abs(model.score_samples(trainX))

plt.hist(score, bins = 50)
plt.show()

# Anomaly score가 1에 가까운 데이터를 이상 데이터로 판정한다.
predY = (score > 0.65).astype(int)
fraud_count = (predY == 1).sum()
print('이상 데이터로 판정한 개수 =', fraud_count)

# confusion matrix를 확인한다.
cm = confusion_matrix(trainY, predY)
cm_df = pd.DataFrame(cm)
cm_df.columns = ['pred_normal', 'pred_abnormal']
cm_df.index = ['actual_normal', 'actual_abnormal']
print(cm_df)

# accuracy는 큰 의미가 없고, precision과 recall이 의미있음.
print('\n비정상으로 판정한 것 중에,')
print('정상 데이터를 비정상으로 판정한 비율 = %.4f' % (cm[0, 1] / cm[:, 1].sum()))
print('비정상 데이터를 비정상으로 판정한 비율 = %.4f' % (cm[1, 1] / cm[:, 1].sum()))
```




```
confusion matrix :
                 pred_normal  pred_abnormal
actual_normal         283865            450
actual_abnormal          345            147

비정상으로 판정한 것 중에,
정상 데이터를 비정상으로 판정한 비율 = 0.7538
비정상 데이터를 비정상으로 판정한 비율 = 0.2462
```