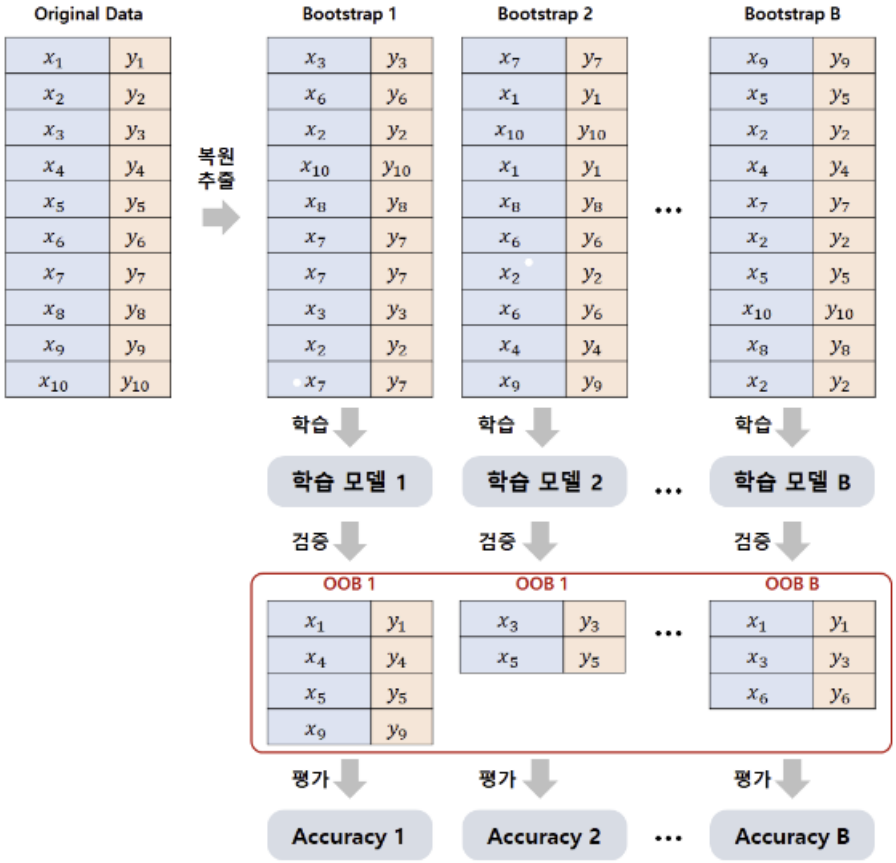
BDA x 이지스 퍼블리싱 머신러닝 스터디 3주차 과제

조 이름: 1조 (김예진, 김지웅, 박효정, 편민우, 홍소은)

---

1. N개의 샘플로 구성된 원 데이터에서 N개의 부트스트래핑 샘플을 생성한다고 해보자. N이 충분히 클 때 OOB 데이터의 비율을 구해보시오.

- 부트스트래핑: 주어진 데이터가 샘플이 아닌 모집단 그 자체라 가정하고 수많은 시뮬레이션 샘플을 만들어내는 기법
- OOB(out-of-ag) 데이터: 부트스트래핑은 복원 추출을 하기 때문에 training set 추출과정에서 원본 데이터셋에는 있으나 부트스트랩 데이터에는 없는 샘플이 발생할 가능성이 높은데, 앙상블 학습 기법의 하나인 배깅(**b**ootstrap **agg**regat**ing**, bagging)에서는 이를 OOB 샘플이라함.



[그림 3] Out of Bag Data

풀이)

부트스트래핑 샘플의 크기가 N이므로, 그 중 하나의 샘플이 뽑히지 않을 확률은 $\frac{N-1}{N}$이고,

이를 N회 복원 추출을 진행했을 경우, 그 샘플이 뽑히지 않을 확률은 $(\frac{N-1}{N})^N$이다.

이때, N의 값이 충분히 크다고 가정하면 어떠한 샘플이 추출되지 않은 확률, 즉 OOB 데이터가 발생할 확률은 $\lim_{n\to\infty}\left(1+\frac{1}{N}\right)^{N} = e^{-1} = 0.368$, 즉 36.8% 이다.

따라서 N이 충분히 클 때 OOB 데이터의 비율은 대략 0.368임을 알 수 있다.

$$\begin{aligned}\lim_{N\to 0}\left(1-\frac{1}{N}\right)^{N} &= \lim_{N\to 0}\left[\left(1-\frac{1}{N}\right)^{-N}\right]^{-1} \\ &= e^{-1} \qquad \left(\because e = \lim_{n\to 0}\left(1+\frac{1}{n}\right)^{n}\right) \\ &= 0.3679\end{aligned}$$

2. 6장 되새김 문제 2번

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split

X, y = load_diabetes(return_X_y = True, as_frame = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1234)
train = pd.concat([X_train, y_train], axis=1)
X_cols, y_col = X.columns.tolist(), y.name

# 문제 조건에 따라 하이퍼파라미터 설정
max_depth = 4 #최대 깊이 설정
min_sample_split = 4 #분할을 수행할 최소 샘플 개수 설정

#MSE를 계산하는 함수
def eval_mse(left, right, y_col):
    mse, n1, n2 = 0, len(left), len(right)

    score = ((left[y_col] - left[y_col].mean())**2).mean()
    mse += score * n1 / (n1 + n2)
    score = ((right[y_col] - right[y_col].mean())**2).mean()
    mse += score * n2 / (n1 + n2)

    return mse

#리프 노드에서의 목푯값 평균값 계산하는 함수
def eval_y(df, y_col):
    return df[y_col].mean()
```

```
#빈 트리 생성
tree = []
for i in range(0, 2**(max_depth + 1)):
    tree.append(dict({'struct': None}))

tree[1]['struct'] = train

for i in range(1, len(tree)):

    #트리 끝난 부분은 통과
    if not isinstance(tree[i]['struct'], pd.DataFrame): continue

    #최대 깊이 도달하면 클래스 출력
    if i >= 2**max_depth:
        tree[i]['struct'] = eval_y(tree[i]['struct'], y_col)
        continue

    data = tree[i]['struct']
    a, b, c, d, e = '', float('inf'), float('inf'), None, None

    #최고의 피처와 임계값을 찾은 후 그 기준으로 분할
    for X_col in X_cols:
        vals = np.sort(data[X_col].unique())
    for val in vals[1:]:
        left, right = data[data[X_col] < val], data[data[X_col] >= val]
        mse = eval_mse(left, right, y_col)
        if mse < c:
            a, b, c, d, e = X_col, val, gini, left, right

    tree[i]['col'] = a
    tree[i]['val'] = b
    if len(d) >= min_sample_split :
        tree[i << 1]['struct'] = d
    else:
        tree[i << 1]['struct'] = eval_y(e, y_col)

    if len(e) >= min_sample_split :
        tree[(i << 1) + 1]['struct'] = e
```

```
    else:
        tree[(i << 1) + 1]['struct'] = eval_y(e, y_col)

error = 0 #error 변수 초기화
#훈련 셋 각 인스턴스에 대해 반복문
for i in range(len(X_train)):
    row = X_train.iloc[i]
    ind = 1 #트리의 루트에서 시작하는 인덱스 초기화
    node = tree[ind]
    #리프 노드에 도달할 때까지 트리 탐색
    while isinstance(node['struct'], pd.DataFrame):
        if row[node['col']] < node['val']: #왼쪽, 오른쪽으로 이동할지 여부 결정
            ind = ind << 1 #왼쪽으로 이동
        else:
            ind = (ind << 1) + 1 #오른쪽으로 이동
            node = tree[ind] #현재 노드를 자식 노드로 업데이트

        y_pred = node['struct'] #현재 인스턴스에 대한 예측값
        error += np.abs(y_pred - y_train.iloc[i]) #오차 업데이트
print(f'학습 데이터셋 MAE: {error / len(y_train): .2f}')

error = 0
for i in range(len(X_test)):
    row = X_test.iloc[i]
    ind = 1
    node = tree[ind]
    while isinstance(node['struct'], pd.DataFrame):
        if row[node['col']] < node['val']:
            ind = ind << 1
        else:
            ind = (ind << 1) + 1
            node = tree[ind]



        y_pred = node['struct']
        error += np.abs(y_pred - y_test.iloc[i])
print(f'테스트 데이터셋 MAE: {error / len(y_test): .2f}')
```

3. 그레이디언트 부스팅 모델은 과적합에 취약하다. 그 이유를 제시하고, 과적합을 방지하는 규제 기법에 대해서 서술해보시오.

1) 그레이디언트 부스팅 모델이 과적합에 취약한 이유

그레이디언트 부스팅 모델((gradient boosting trees, GBT)은 부스팅 기법의 하나인 GBM을 결정 트리에 적용한 앙상블 모델로, 모델이 예측을 하면 그 에측과 실제값 차이인 잔차가 발생하는데 GBM은 반복 학습을 통해 잔차를 최소화하는 방향으로 새로운 모델을 생성한다. Greedy Algorithm을 주로 사용, 이는 미래를 생각하지 않고 각 단계에서 가장 최선의 선택을 하는 기법) 모델의 학습 과정에서 GBM은 이전 머신러닝이 학습하지 못한 잔차에 초점을 두기 때문에 불필요한 모집단의 오차까지 학습이 되고 과적합이 발생하게 된다.

2) 과적합을 방지하는 규제 기법

2-1) Subsampling

- 모집단에서 일부 샘플링한 데이터로 모델 학습
- 복원 추출과 비복원 추출 모두 사용 가능

2-2) Shrinkage

- 뒤쪽에 생성된 모델의 가중치를 줄여 모델에게 주는 영향력을 감소시키는 가장 최신 방법

$$y = f0(x) + f1(x) + f2(x) + \ldots + 0.9 * fn-2(x) + 0.8 * fn-1(x) + 0.7 * fn(x)$$

위와 같이 가장 최근에 생성된 모델의 가중치를 점점 줄이는 방법이 shrinkage 방법입니다.

2-3) Early Stopping

- 머신러닝 학습 과정에서 학습 횟수를 사전 설정할 때, 에러 감소 등 성능이 개선되는지 관찰하는 감시자 역할
- 감시하다 성능의 큰 변화가 없을 시, 설정한 학습 횟수에 도달하지 못하더라도 학습을 중단시켜 과적합 방지함

2-4) Variable importance

- 랜덤 포레스트와 마찬가지로, 변수의 중요도를 측정할 수 있음.
- 하나의 의사 결정 나무 T에서 변수 j의 중요도는 해당 변수를 사용했을 때 얻어지는 정보 획득량(information gain, IG)을 모두 더하여 측정

$$\text{Influence}_j(T) = \sum_{i=1}^{L-1}(IG_i \times \mathbf{1}(S_i = j))$$

- 식에서 L은 해당 트리의 리프 노드의 개수이며 분기 횟수는 리프 노드 개수보다 1개 적기 때문에, I = 1, ..., L-1이 되고, I(Si=j)은 지시 함수(Indicator function)로 뒤 조건이 일치할 때는 1, 그렇지 않는다면 0의 값을 가짐. 여기서 i번째 분기에 사용된 변수 Si가 j와 동일할 때만 1의 값을 나타냄.
- 그래디언트 부스팅 머신에서 변수의 중요도는 각 트리마다 변수 j의 중요도를 평균 내어 구함

$$\text{Influence}_j = \frac{1}{M}\sum_{k=1}^{M}\text{Influence}_j(T_k)$$

4. XGBoost, LightGBM, CatBoost 모델의 특징을 서술하시오.

4-1) XGBoost(Extreme Gradient Boost)

- GBT 모델에 병렬 처리, 하드웨어 최적화, 과적합 규제 페널티 등의 여러 개념을 도입하여 최적화한 모델
    - 손실 함수가 최대한 감소하도록 하는 split
    - point(분할점)을 찾는 모델
    - 과적합 규제 기능(기존 GBT에선 없었음)
    - Early Stopping 기능
- 다양한 파라미터와 Customizing 용이
- 근사법 사용하기 때문에 편향성은 일부 증가하지만, 시간 복잡도는 크게 낮아짐

- 최신 모델 중 상대적으로 느린 편, 파라미터 튜닝 시 시간이 더욱 오래 걸림
- 학습률이 높을수록 과적합되기 쉬움
- 다른 앙상블 계열 알고리즘과 같이 해석이 어려움

4-2) LightGBM

- XGBoost의 장점은 끌어오고 단점은 보완하는 방식으로 개발
- 학습에 걸리는 시간과 메모리 사용량 줄임 (큰 사이즈의 데이터를 다룰 수 있는데 적은 메모리 차지)
- 더 나은 정확도, 결과의 정확도에 초점 맞춤
- 병렬, 분산 및 GPU 학습 지원
- 리프 중심의 분할 트리 방식 사용
- 가벼운 GBT 모델 제공

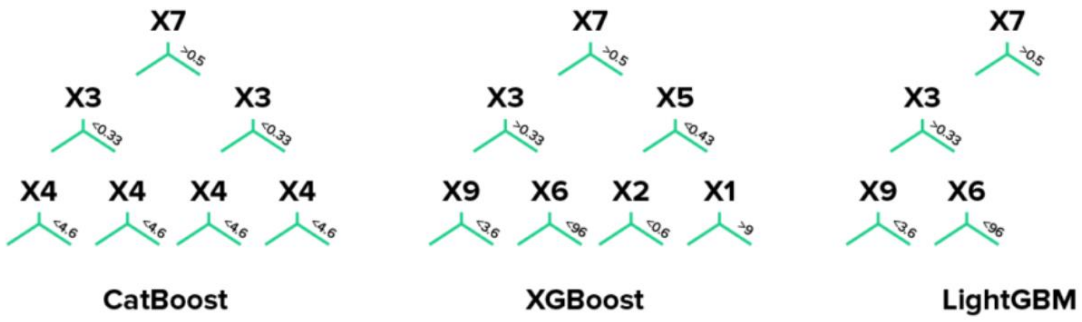
- 과적합에 민감하고 작은 데이터에 대해 과적합되기 쉬움 (적은 데이터에는 비추천)
- 구현은 쉬우나 파라미터 튜닝 복잡함

4-3) CatBoost

- 앙상블 기법 중 하나 (Boosting류 중 하나)
- 기존의 부스팅 모델은 모든 훈련 데이터를 대상으로 잔차를 계산하였으나, CatBoost는 학습 데이터의 일부로 잔차 계산을 한 뒤, 해당 결과로 모델을 재생성
- 10행의 데이터가 있다면 특정 2행만 학습하고 다음 4행 학습, 이런 식으로 진행: 'Ordered Boosting' 이라 함
- 모델 및 기능 분석을 위한 시각화 및 도구
- 시계열 데이터를 효율적으로 처리 가능
- 범주형 기능에 대한 기본 처리
- 빠른 GPU 훈련으로 속도가 매우 빠름 (XGBoost보다 약 8배 빠른 것으로 알려짐)
- 오버 피팅을 막기 위한 여러 방법(random permutation, overfitting detector)을 갖춰 예측력 높음
- 하이퍼 파라미터 튜닝 지정하지 않아도 모델이 최적화되어 잘 돌아감

- 수평 트리 대칭으로 나누어지는 것이 특징

- 수치형 데이터가 많을 때 상대적으로 훈련 시간이 김
- 결측치가 매우 많은 데이터셋에는 부적합함



Function	XGBoost	CatBoost	Light GBM
Important parameters which control overfitting	1. **learning_rate or eta** – optimal values lie between 0.01-0.2 2. **max_depth** 3. **min_child_weight:** similar to min_child leaf; default is 1	1. **Learning_rate** 2. **Depth** - value can be any integer up to 16. Recommended - [1 to 10] 3. No such feature like min_child_weight 4. **l2-leaf-reg**: L2 regularization coefficient. Used for leaf value calculation (any positive integer allowed)	1. **learning_rate** 2. **max_depth**: default is 20. Important to note that tree still grows leaf-wise. Hence it is important to tune **num_leaves** (number of leaves in a tree) which should be smaller than 2^(max_depth). It is a very important parameter for LGBM 3. **min_data_in_leaf**: default=20, alias= min_data, min_child_samples
Parameters for categorical values	Not Available	1. **cat_features**: It denotes the index of categorical features 2. **one_hot_max_size**: Use one-hot encoding for all features with number of different values less than or equal to the given parameter value (max – 255)	1. **categorical_feature:** specify the categorical features we want to use for training our model
Parameters for controlling speed	1. **colsample_bytree**: subsample ratio of columns 2. **subsample**: subsample ratio of the training instance 3. **n_estimators**: maximum number of decision trees; high value can lead to overfitting	1. **rsm:** Random subspace method. The percentage of features to use at each split selection 2. No such parameter to subset data 3. **iterations**: maximum number of trees that can be built; high value can lead to overfitting	1. **feature_fraction:** fraction of features to be taken for each iteration 2. **bagging_fraction:** data to be used for each iteration and is generally used to speed up the training and avoid overfitting 3. **num_iterations:** number of boosting iterations to be performed; default=100

5. 7장 되새김 문제 3번

```
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split

# 당뇨병 데이터 불러오기
diabetes = load_diabetes(as_frame = True)
# 데이터셋 분리
X_train, X_val, y_train, y_val = train_test_split(diabetes.data, diabetes.target, random_state=0)

from sklearn.linear_model import Ridge
from sklearn.inspection import permutation_importance
# 릿지 회귀 모델 훈련
model = Ridge(alpha=1e-2).fit(X_train, y_train)
# 퍼뮤테이션 중요도 계산
pi = permutation_importance(model, X_val, y_val, n_repeats=30, random_state=0, scoring='neg_mean_squared_error')

# 중요도 높은 상위 3개 피처 출력
pi_series = pd.Series(pi.importances_mean, index=X_train.columns)
print(f'퍼뮤테이션 기반 피처별 중요되 {pi_series.sort_values(ascending=False).values[:3]}')
```

```
퍼뮤테이션 기반 피처별 중요되 [1013.86634639  872.72567747  438.66275116]
```

6. 8장 되새김 문제 2번

```
import pandas as pd
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.feature_selection import RFE

X, y = make_classification(n_samples=300, n_features=100, n_informative=30, n_redundant=15, n_repeated=5, n_classes=2, flip_y=0.05, random_state=1234)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1234)

clf = GradientBoostingClassifier(random_state=1234)
y_pred = clf.fit(X_train, y_train).predict(X_test)
```

```
print(f'전체 피처를 사용한 GBT 모델의 정확도: {np.mean(y_pred == y_test) * 100:.2f}%')
```

전체 피처를 사용한 GBT 모델의 정확도: 66.67%

```
clf = GradientBoostingClassifier(random_state=1234)
selector = RFE(clf, n_features_to_select = 20, step = 1)
selector = selector.fit(X_train, y_train)
selector.support[:10]
```

```
Array([False, False, False, True, False, True, True, False, False,
False])
```

```
X_train2 = X_train.iloc[:, selector.support_]
X_test2 = X_test.iloc[:, selector.support_]

clf = GradientBoostingClassifier(random_state=1234)
y_pred = clf.fit(X_train2, y_train).predict(X_test2)

print(f'RFE 클래스 기반 후진 소거법을 적용한 GBT 모델의 정확도: {(y_pred == y_test).mean() *
100:.2f}%')
```

RFE 클래스 기반 후진 소거법을 적용한 GBT 모델의 정확도: 75.76%