# 4주차 예습과제

## 4.5 GBM의 개요 및 실습

- 부스팅 알고리즘: 여러 개의 약한 학습기를 순차적으로 학습-예습, 잘못 예측한 데이터에 가중치 부여해 오류를 개선해 나가면서 학습
  - 에이다 부스트(AdaBoost): 오류 데이터에 가중치를 부여하면서 부스팅 수행, 여러 학습기를 가중 평균하여 결합함
  - 그래디언트 부스트(GBM): 가중치 업데이트를 경사 하강법을 이용
  - 경사 하강법: h(x)=y-F(x)를 최소화하는 방향으로 가중치 값을 업데이트

---

- 사이킷런은 GBM 기반의 분류를 위해 GradientBoostingClassifier를 제공한다

```
gb_clf=GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train,y_train)
gb_pred=gb_clf.predict(X_test)
gb_accuracy=accuracy_score(y_test,gb_pred)
```

- 일반적으로 GBM이 랜덤 포레스트보다 예측 성능이 뛰어나나 수행 시간이 오래 걸리고 하이퍼 파라미터 튜닝도 복잡하다

---

**[하이퍼 파라미터]**

- loss: 경사 하강법에서 사용할 비용 함수 (default=deviance)
- learning_rate: 학습률, 학습기가 순차적으로 오류 값을 보정해 나가는데 적용하는 계수(default=0.1) → n_estimator와 상호 보완적으로 조합해 사용함
- n_estimators: weak learner의 개수(default=100)
- subsample: 학습에 사용하는 데이터의 샘플링 비율(default=1, 전체 학습 데이터 기반으로 학습)

## 4.6 XGBoost(eXtra Gradient Boost)

- GBM에 기반하며 느린 수행 시간 및 과적합 규제 부재 등의 문제를 해결함

**[장점]**

| 항목 | 설명 |
| --- | --- |
| 뛰어난 예측 성능 | 일반적으로 분류와 회귀 영역에서 뛰어난 예측 성능을 발휘합니다. |
| GBM 대비 빠른 수행 시간 | 일반적인 GBM은 순차적으로 Weak learner가 가중치를 증감하는 방법으로 학습하기 때문에 전반적으로 속도가 느립니다. 하지만 XGBoost는 병렬 수행 및 다양한 기능으로 GBM에 비해 빠른 수행 성능을 보장합니다. 아쉽게도 XGBoost가 일반적인 GBM에 비해 수행 시간이 빠르다는 것이지, 다른 머신러닝 알고리즘(예를 들어 랜덤 포레스트)에 비해서 빠르다는 의미는 아닙니다. |
| 과적합 규제 (Regularization) | 표준 GBM의 경우 과적합 규제 기능이 없으나 XGBoost는 자체에 과적합 규제 기능으로 과적합에 좀 더 강한 내구성을 가질 수 있습니다. |
| Tree pruning (나무 가지치기) | 일반적으로 GBM은 분할 시 부정 손실이 발생하면 분할을 더 이상 수행하지 않지만, 이러한 방식도 자칫 지나치게 많은 분할을 발생할 수 있습니다.<br>다른 GBM과 마찬가지로 XGBoost도 max_depth 파라미터로 분할 깊이를 조정하기도 하지만, tree pruning으로 더 이상 긍정 이득이 없는 분할을 가지치기 해서 분할 수를 더 줄이는 추가적인 장점을 가지고 있습니다. |
| 자체 내장된 교차 검증 | XGBoost는 반복 수행 시마다 내부적으로 학습 데이터 세트와 평가 데이터 세트에 대한 교차 검증을 수행해 최적화된 반복 수행 횟수를 가질 수 있습니다.<br>지정된 반복 횟수가 아니라 교차 검증을 통해 평가 데이터 세트의 평가 값이 최적화 되면 반복을 중간에 멈출 수 있는 조기 중단 기능이 있습니다. |
| 결손값 자체 처리 | XGBoost는 결손값을 자체 처리할 수 있는 기능을 가지고 있습니다. |

- xgboost=XGBoost 전용의 파이썬 패키지 + 사이킷런과 호환되는 래퍼용 XGBoost
- xgboost 패키지의 사이킷런 래퍼 클래스: XGBClassifier + XGBRegressor

---

**[하이퍼 파라미터]**

: GBM과 유사한 파라미터 + 조기중단, 과적합 규제를 위한 하이퍼 파라미터

*! 파이썬 래퍼 XGBoost 모듈과 사이킷런 래퍼 XGBoost모듈의 일부 하이퍼 파라미터는 다름 !*

- 일반 파라미터: 디폴트 파라미터 값을 바꾸는 경우가 거의 없음
  - booster:
    gbtree(tree based model) / gblinear(liner model) 중 선택
    (default=gbtree)
  - silent:
    default=0, 출력 메세지 나타내고 싶지 않으면 1로 설정
  - nthread:
    CPU의 실행 스레드 개수 조정 (default=전체 스레드 사용)
- 부스터 파라미터: 트리 최적화, 부스팅, regularization 등과 관련된 파라미터
  - eta (learning_rate):
    GBM의 learning_rate와 같은 것.  (default=0.3)

    - num_boost_rounds:
      GBM의 n_estimators와 같은 것
    - min_child_weight:
      트리에서 가지 나뉠 때 필요한 데이터 weight 총합 → 클수록 과적합 방지 (default=1)
    - gamma(min_split_loss):
      트리에서 가지 나뉠지 결정하는 최소 손실 감소 값 → 클수록 과적합 방지 (default=0)
    - max_depth: 트리 기반 알고리즘의 max_depth와 같음. → 작을수록 과적합 방지 (default=6)
    - subsample:
      GBM의 subsample과 같은 것. 각 트리 학습 시 샘플링 비율 (default=1)
    - colsample_bytree:
      GBM의 max_features와 같은 것. 트리 생성 시 사용할 feature 비율 (default=1)
    - lambda (reg_lambda):
      L2 Regularization 항. 피처 가중치에 대한 L2 규제 (default=1) → 클수록 과적합 방지
    - alpha (reg_alpha):
      L1 Regularization 항. 피처 가중치에 대한 L1 규제 (default=0)
    - scale_pos_weight:
      불균형 데이터(비대칭 클래스)에서 특정 값으로 가중치를 조정해 균형을 맞추는 파라미터 (default=1)
- 학습 태스크 파라미터: XGBoost가 예측 시 사용하는 목적 함수 및 평가 지표를 설정하는 파라미터
    - objective: 최소값을 가져야 할 손실 함수를 정의. 이진/다중 분류 여부에 따라 다른 손실함수 사용
        - binary:logistic:  이진 분류 시 사용
        - multi:softmax:  다중 분류 시 사용. 손실함수가 multi:softmax일 경우 num_class(클래스 개수) 파라미터 지정 필요
        - multi:softprob:  multi:softmax와 유사하나 각 클래스별 예측 확률을 반환
    - eval_metric: 검증에 사용할 평가 지표 설정. (default= 회귀→rmse, 분류→error)

        - rmse: Root Mean Square Error
        - mae: Mean Absolute Error
        - logloss: Negative log-likelihood
        - error: Binary classification error rate (0.5 threshold)
        - merror: Multiclass classification error rate
        - mlogloss: Multiclass logloss
        - auc: Area Under the Curve
- 과적합 문제 해결 방법
    1. eta 값을 낮추고(0.01~0.1) num_boost_rounds를 높여준다
    2. max_depth 값을 낮춘다
    3. min_child_weight 값을 높인다
    4. gamma 값을 높인다
    5. subsample과 colsample_bytree를 조정한다
- xgboost 패키지는 자체적으로 교차 검증, 성능 평가, 피처 중요도 등의 시각화 기능을 가지고 있다.
- 조기 중단 기능: 지정한 부스팅 반복 횟수에 도달하지 않더라도 예측 오류가 더 이상 개선되지 않으면 반복을 중지해 수행 시간을 개선한다

---

## 파이썬 래퍼 XGBoost

```
import xgboost as xgb
xgb_model=xgb.train()
pred_probs=xgb_model.predict(dtest)
preds=[1 if x>0.5 else 0 for x in pred_probs]
```

- XGBoost만의 전용 데이터 객체인 **DMatrix**를 사용한다
    - **xgb.DMatrix**(data=피처데이터세트,label=레이블데이터세트) → DMatrix로 변환
- train(): 학습이 완료된 모델 객체 반환
    - 하이퍼 파라미터를 train()함수에 입력한다
    - XGBoost의 하이퍼 파라미터는 주로 딕셔너리 형태로 입력

    - early_stopping_rounds: 조기 중단 할 수 있는 최소 반복 횟수 지정해주는 파라미터
    - evals: 평가용 데이터 세트 지정 → [(dval,'**eval**')] 처럼 튜플 형태로 입력
    - eval_metric: 지정된 평가용 데이터 세트에서 eval_metric에 지정된 평가 지표로 예측 오류를 측정한다
- predict()는 예측 결괏값이 아닌 확률 값을 반환한다
- plot_importance: f스코어를 기반으로 피처의 중요도를 막대그래프로 나타냄
- xgboost.to_graphviz( ): 규칙 트리 구조 그릴 수 있다.
- xgboost.cv(): GridSearchCV와 유사하게 최적 파라미터 구함
    - params (dict): 부스터 파라미터
    - dtrain (DMatrix): 학습 데이터
    - num_boost_round (int): 부스팅 반복 횟수
    - nfold (int): CV 폴드 개수
    - stratified (bool): CV 수행 시 층화 표본 추출(stratified sampling) 수행 여부
    - metrics (string or list of strings): CV 수행 시 모니터링할 성능 평가 지표
    - early_stopping_rounds (int): 조기 중단을 활성화시킴. 반복 횟수 지정

## 사이킷런 래퍼 XGBoost

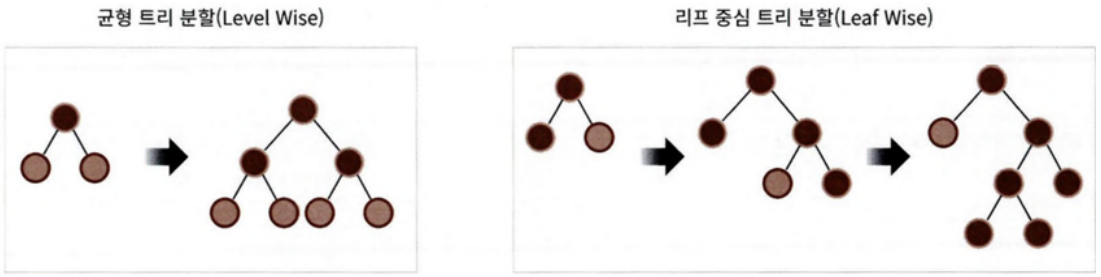
```
from xgboost import XGBClassifier
xgb_wrapper=XGBClassifier()
xgb_wrapper.fit(X_train,y_train,verbose=True)
w_preds=xgb_wrapper.predict(X_test)
```

- 하이퍼 파라미터에 차이가 있다
    - eta → learning_rate
    - sub_sample → subsample
    - lambda → reg_lambda
    - alpha → reg_alpha
- 조기중단: 관련 파라미터를 fit()에 입력
    - early_stopping_rounds

    - eval_metric
    - eval_set → 앞 튜플이 자동으로 학습용 데이터, 뒤 튜플이 검증용 데이터로 인식됨
- plot_importance(): 똑같이 피쳐 중요도 시각화해줌

# 4.7 LightGBM

- XGBoost보다 학습이 빠르고 메모리 사용량도 상대적으로 적은데 예측 성능은 차이가 없다.
- 카테고리형 피처의 자동 변환과 최적 분할 가능 (원-핫 인코딩 등이 필요X)
- 하지만 적은 데이터 세트에 적용할 경우 과적합이 발생하기 쉽다 (일반적으로 10,000건 이하)
- 리프 중심 트리 분할 방식을 사용 → 트리의 균형을 맞추지 않고 최대 손실값을 가지는 리프 노드를 계속 분할하면서 비대칭적인 규칙 트리가 생성됨




**[하이퍼 파라미터]**

: XGBoost와 유사하지만 리프 노드가 계속 분할되면서 트리가 깊어지므로 이에 맞는 하이퍼 파라미터 설정이 필요

- num_iterations (default=100) : 트리 반복 횟수. 너무 크면 과적합, 너무 작으면 성능 저하.
- learning_rate (default=0.1) : 학습률. 작게 하면 학습 시간이 길어지고, 과적합 방지 가능.
- max_depth (default=-1) : 트리 깊이 제한. -1이면 제한 없음.
- min_data_in_leaf (default=20) : 리프 노드 최소 데이터 수. 작게 설정하면 과적합 가능.
- num_leaves (default=31) : 최대 리프 수.
- boosting (default='gbdt') : 부스팅 유형. gbdt(기본), rf(랜덤 포레스트) 등.
- bagging_fraction (default=1.0) : 데이터 샘플링 비율. 과적합 방지용.

- feature_fraction (default=1.0) : 학습에 사용될 피처 비율. 과적합 방지용.
- lambda_l2 (default=0.0) : L2 규제(과적합 제어).
- lambda_l1 (default=0.0) : L1 규제(과적합 제어).
- objective : 최적화할 손실함수 지정. 분류/회귀 등 task에 따라 다름.

| 유형 | 파이썬 래퍼 LightGBM | 사이킷런 래퍼 LightGBM | 사이킷런 래퍼 XGBoost |
|---|---|---|---|
| 파라미터명 | num_iterations | n_estimators | n_estimators |
| | learning_rate | learning_rate | learning_rate |
| | max_depth | max_depth | max_depth |
| | min_data_in_leaf | min_child_samples | N/A |
| | bagging_fraction | subsample | subsample |
| | feature_fraction | colsample_bytree | colsample_bytree |
| | lambda_l2 | reg_lambda | reg_lambda |
| | lambda_l1 | reg_alpha | reg_alpha |
| | early_stopping_round | early_stopping_rounds | early_stopping_rounds |
| | num_leaves | num_leaves | N/A |
| | min_sum_hessian_in_leaf | min_child_weight | min_child_weight |

- 기본 튜닝 방안: numleaves의 개수를 중심으로 min_child_samples (min_data_in_leaf), max_depth를 함께 조정하면서 모델의 복잡도를 줄이는 것
- learning_rate를 작게 하면서 n_estimators를 크게 하는 것도 적용 가능
- reg_lambda, reg_alpha같은 regularization적용
- 학습 데이터에 사용할 피처의 개수나 데이터 샘플링 레코드 개수를 줄이기 위해 colsample_bytree, subsample 파라미터를 적용

## 4.8 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

- Grid Search는 튜닝해야 할 하이퍼 파라미터 개수가 많을 경우 최적화 수행 시간이 오래 걸림
- XGBoost나 LightBGM은 하이퍼 파라미터 개수가 많기 때문에 Grid Search 방법은 부적합

⇒ 대용량 학습 데이터에 XGBoost나 LightGBM의 하이퍼 파라미터 튜닝 시 베이지안 최적화 기법 등의 다른 방식을 적용한다.

- 베이지안 최적화: 블랙 박스 형태의 함수에서 최대 또는 최소 반환 값을 만드는 최적 입력값을 빠르고 효과적으로 찾아주는 방식
  - 대체 모델: 획득 함수로부터 추천받은 입력값(하이퍼파라미터)을 이용해 최적 함수 모델을 점차 개선 → 일반적으로 가우시안 프로세스를 적용
  - 획득 함수: 개선된 대체 모델을 기반으로 최적 입력값(하이퍼파라미터)을 계산하고 추천

1. 랜덤하게 하이퍼 파라미터들을 샘플링하고 성능 결과를 관측
2. 관측된 값을 기반으로 대체 모델은 최적 함수를 추정
3. 추정된 최적 함수를 기반으로 획득함수는 다음으로 관측할 하이퍼 파라미터 값을 계산 → 이전 최적 관측값보다 더 큰 최댓값을 가질 가능성이 높은 지점을 찾음 (=하이퍼 파라미터)
4. 이를 수행해서 관측된 값을 기반으로 대체 모델은 갱신되고 다시 최적 함수를 예측 추정함

⇒ step 3와 4를 반복함

---

- 베이지안 최적화를 제공하는 파이썬 패키지로는 HyperOpt, Bayesian Optimization, Optuna 등이 있다

1. 입력 변수명과 입력값의 검색 공간 설정
   - hp.quniform(label, low, high, q): 지정된 입력값 변수를 low~high 범위에서 q 간격으로 탐색.
   - hp.uniform(label, low, high): low~high 범위에서 연속 균등 분포 형태로 탐색.
   - hp.randint(label, upper): 0부터 upper까지 무작위 정수로 탐색.
   - hp.loguniform(label, low, high): exp(uniform(low, high)) 값을 반환하며, 로그 변환된 값이 균등 분포 형태로 탐색.
   - hp.choice(label, options): 문자열 또는 숫자 등 리스트/튜플 형태의 값 중 하나를 선택하여 탐색.
2. 목적 함수 설정
3. 목적 함수의 반환 **최솟값**을 가지는 최적 입력값 유추 → **fmin()**
   - fn: 최소화하려는 목적 함수.

- space: 최적화할 하이퍼 파라미터들의 탐색 검색 공간.
- algo: 최적화를 수행할 알고리즘 (기본값: TPE).
- max_evals: 최적의 조합을 찾기 위한 최대 반복 횟수.
- trials: 탐색 과정의 모든 시도와 결과 이력을 저장하는 객체.
  - Trials 객체: results/vals 속성을 가짐
- rstate: 매번 동일한 결과를 얻기 위한 랜덤 시드 값.

---

- HyperOpt를 이용하여 XGBoost의 하이퍼 파라미터를 최적화:
  1. 적용해야 할 하이퍼 파라미터와 검색 공간을 설정
  2. 목적 함수에서 XGBoost를 학습
  3. 예측 성능 결과를 반환 값으로 설정
  4. fmin() 함수에서 목적 함수를 하이퍼 파라미터 검색 공간의 입력값들을 사용하여 최적의 예측 성능 결과를 반환하는 최적 입력값들을 결정
- HyperOpt는 입력값과 반환 값이 모두 실수형이기 때문에 하이퍼 파라미터 입력 시 형 변환을 해줘야 한다.
- 성능 값이 클수록 좋은 성능 지표일 경우 -1을 곱해 줘야 한다.

## 4.10 분류 실습-캐글 신용카드 사기 검출

- 레이블이 불균형하면 예측 성능에 문제가 발생할 수 있다
  - 오버 샘플링: 적은 레이블을 가진 데이터 세트를 많은 레이블을 가진 데이터 세트 수준으로 증식
  - 언더 샘플링: 많은 레이블을 가진 데이터 세트를 적은 레이블을 가진 데이터 세트 수준으로 감소 → 원본 데이터의 피처 값들을 아주 약간 변경하여 증식
    - SMOTE 방법: 개별 데이터들의 K 최근접 이웃을 찾아서 이 데이터와 K개 이웃들의 차이를 일정 값으로 만들어서 기존 데이터와 약간 차이가 나는 새로운 데이터들을 생성 ⇒ imbalanced-learn 패키지
    - 재현율은 높아지나 정밀도는 낮아지는 것이 일반적임.

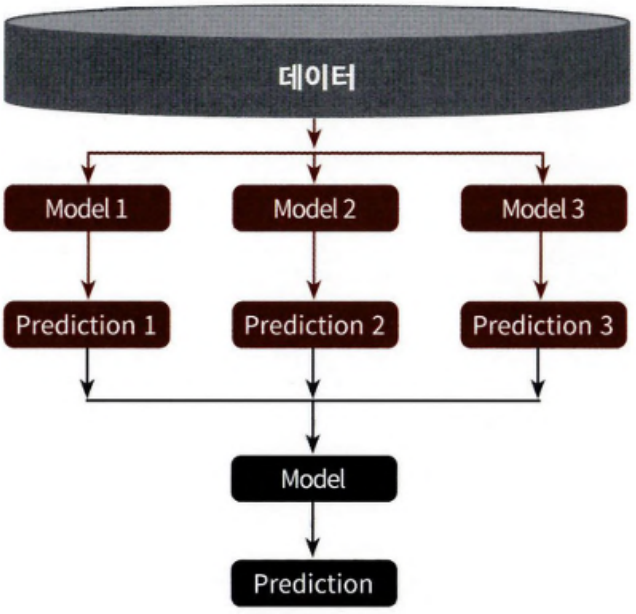
```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=0)
X_train_over, y_train_over = smote.fit_resample(X_train, y_train)
```

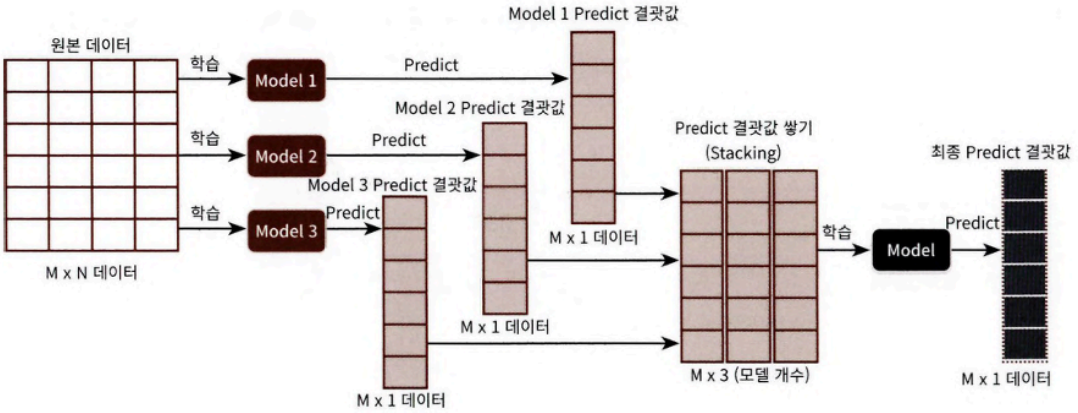
- 이상치 데이터: 전체 데이터의 패턴에서 벗어난 이상 값을 가진 데이터
  - IQR: 사분위(Quantile) 값의 편차를 이용해 이상치를 찾아내는 방법
  - 25% 구간인 Q1 ~ 75% 구간인 Q3의 범위
  - IQR에 1.5를 곱해서 생성된 범위를 이용해 최댓값과 최솟값을 결정한 뒤 최댓값을 초과하거나 최솟값에 미달하는 데이터를 이상치로 간주

```
def get_outlier(df=None, column=None, weight=1.5):
    fraud = df[df['Class']==1][column]
    quantile_25 = np.percentile(fraud.values, 25)
    quantile_75 = np.percentile(fraud.values, 75)
    iqr = quantile_75-quantile_25
    iqr_weight = iqr * weight
    lowest_val = quantile_25-iqr_weight
    highest_val = quantile_75 + iqr_weight
    outlier_index = fraud[(fraud < lowest_val)|(fraud > highest_val)].index
    return outlier_index
```

## 4.11 스태킹 앙상블

- 스태킹: 개별적인 **여러 알고리즘의 예측 결과** 데이터 세트를 최종적인 **메타 데이터 세트**로 만들어 별도의 ML 알고리즘으로 최종 학습을 수행하고 테스트 데이터를 기반으로 다시 최종 예측을 수행






## CV 세트 기반의 스태킹

- **과적합을 개선**하기 위해 최종 메타 모델을 위한 데이터 세트를 만들 때 **교차 검증 기반**으로 예측된 결과 데이터 세트를 이용

1. 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터를 생성
2. 개별 모델들이 생성한 학습용 / 테스트용 데이터를 모두 스태킹 형태로 합쳐서 메타 모델이 학습할 최종 학습용 데이터 세트를 생성
3. 최종적으로 생성된 학습 데이터 세트와 원본 학습 데이터의 레이블 데이터를 기반으로 학습한 뒤, 최종적으로 생성된 테스트 데이터 세트를 예측하고, 원본 테스트 데이터의 레이블 데이터를 기반으로 평가

- concatenate(): 여러 넘파이 배열을 칼럼 또는 로우 레벨로 합쳐줌
- 스태킹을 이루는 개별 알고리즘 모델은 최적으로 파라미터를 튜닝한 상태에서 스태킹 모델을 만드는 것이 일반적임