4주차_4장. 분류 - Part2(4.5장 ~ 4.8장, 4.10장 ~ 4.11장)

상태	진행 중

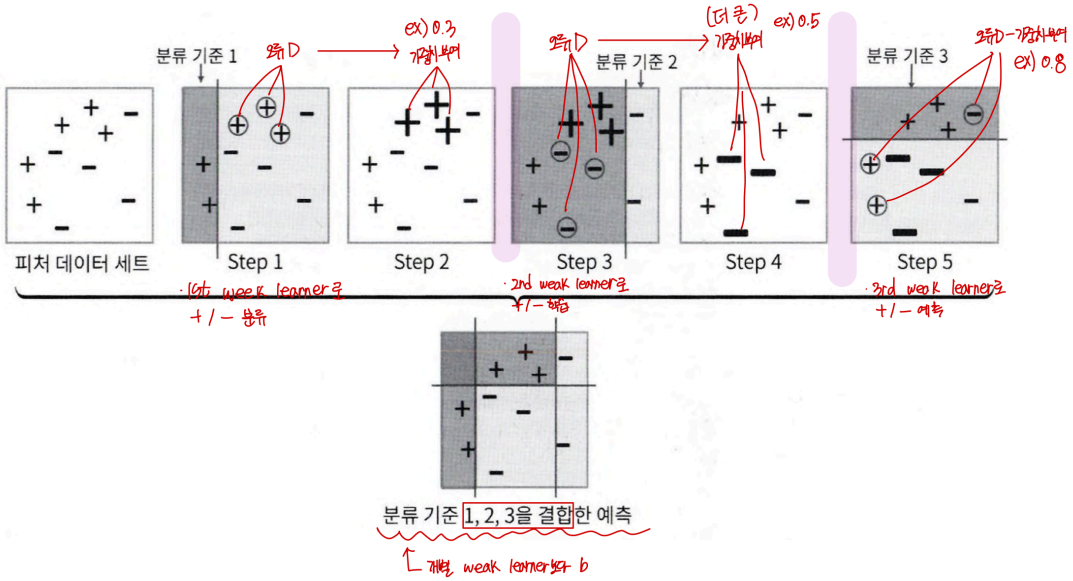
4. 분류(Classification)

4.5 GBM(Gradient Boosting MachiGBMne)

GBM의 개요 및 실습

- 부스팅 : 여러 약한 알고리즘을 순차적으로 학습-예측하며, 잘못된 예측에 가중치 부여로 오류를 개선하는 학습 방식

▼ AdaBoost



- 개별 weak learner에 각각 가중치 부여한 후 모두 결합해 예측 수행

▼ Gradient Boost Machine(GBM)

- 가중치 업데이트를 Gradient Descent(경사하강법)로 이용
- 오류 값

> 💡 y - ŷ / (실제값) - (예측값)

- 오류식

> 💡 h(x) = y - F(x)

- 실제 값 y
- 피처 x → 피처 예측 함수 F(x)

이 **오류식을 최소화하는 방향성**을 가지고, 반복적으로 가중치 값 update

- 실제 예측

```
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)

print('GBM 정확도 : {0:.4f}'.format(gb_accuracy))
print('GBM 정확도 : {0:.1f}초'.format(time.time() - start_time))
```

```
GBM 정확도 : 0.9389
GBM 수행 시간 : 537.7초
```

  - (+)일반적으로 앞 랜덤 포레스트보다는 나은 예측 성능(93.89%)
  - (-) 수행 시간 ↑, 하이퍼 파라미터 튜닝 필요

GBM 하이퍼 파라미터 소개

- `loss` : gradient descent에서 사용할 비용 함수 지정 (Default : '`deviance`')
- `learning_rate` : 학습을 진행할 때마다 적용하는 학습률
  - 0~1 사이 값 (Default : 0.1)
  - 너무 작은 값 : (+)업데이트 값 ↓ → 최소 오류 값을 찾아 성능 ↑

    (-)수행 시간 ↑, 모든 weak learner 반복 완료돼도 최소 오류 값 찾지 X
  - 너무 큰 값 : (+) 수행 시간 ↓

    (-)최소 오류 값 찾지 못하고 지나쳐 성능 ↓

  → `n_estimators`**와 상호 보완적으로 조합!**

  `learning_rate` ↓, `n_estimators` ↑ : 성능 한계점까지는 예측 성능이 조금씩 좋아짐

  ↔ 수행 시간 ↑, 성능이 현격히 좋아지는 수준은 X
- `n_estimators` : weak learner의 개수 (Default : 100)
  - 많을수록 순차적으로 오류를 보정해 일정 수준까지는 성능 ↑ ↔ 수행시간 ↑

- `subsample` : weak learner가 학습에 사용하는 데이터의 샘플링 비율 (Default : 1 = 학습 데이터의 100%를 기반으로 학습)
    - 과적합 염려 시 1 미만으로 설정

4.6 XGBoost(exTra Gradient Boost)

XGBoost 개요

- **트리 기반** 앙상블 학습에서 유명
- GBM에 기반하지만, ↔ 수행시간 ↑, 과적합 규제 부재 문제 해결!
- 병렬 CPU 환경에서 병렬 학습 可, 속도 ↑

▼ 장점

1. 뛰어난 예측 성능
    - 분류/회귀서 뛰어남
2. GBM 대비 빠름
    - 순차적 GBM ↔ 병렬 XGBoost
3. 과적합 규제 Regularization
    - 자체 과적합 규제 기능
4. Tree pruning (가지치기)
    - GBM도 분할 시 부정 손실 발생하면 분할 중지하지만, 그래도 과잉 분할 발생 가능
    - **더 이상 긍정 이득이 없는 분할**을 **가지치기** + `math_depth` 파라미터 → 분할 수 더 줄임
5. 자체 내장된 교차검증
    - 반복 때마다 train set과 test set 에 대해 교차 검증 수행 → 최적화된 반복 수행 횟수 가짐
    - 평가 값이 최적화되면 지정 반복 횟수 전 반복 중지 可 = **조기 중단 기능**
6. 결손값 자체 처리

- 파이썬 패키지명 : `xgboost`
    - 내부에 XGBoost 전용 파이썬 패키지와 사이킷런 호환 래퍼용 XGBoost 존재

파이썬 래퍼 XGBoost 하이퍼 파라미터

1. 일반 파라미터	2. 부스터 파라미터	3. 학습 테스크 파라미터
• 스레드 개수나 silent 모드 등의 선택을 위한 파라미터 • 디폴트 값을 바꾸는 경우 거의 X	• 트리 최적화, 부스팅, Regularization 등 관련 파라미터 지칭 • 대부분의 파라미터가 속함	• 학습 수행 시의 객체 함수 / 평가를 위한 지표 등을 설정 파라미터

▼ 일반 파라미터

1. `booster` : gbtree(tree based model) 또는 gblinear(linear model) 선택. (Default : gbtree)
2. `silent` : 출력 메시지 나타내고 싶지 않을 때 1. (Default : 0)
3. `nthread` : CPU의 실행 스레드 개수 조정. (Default : CPU의 전체 스레드 모두 사용)

▼ 부스터 파라미터

1. `eta [default=0.3, alias: learning_rate]` : GBM의 learning rate와 동일. 0~1 사이 값 지정
2. `num_boost_rounds` : GBM의 `n_estimators`와 동일
3. `min_child_weight[default=1]` : 트리에서 추가적으로 데이터를 나눌지 결정하기 위해 필요한 D들의 weight 총합. 클수록 분할 자제
4. `gamma [default=0, alias: min_split_loss]` : 트리의 리프 노드를 추가적으로 나눌지를 결정할 최소 손실 감소 값. 해당 값보다 큰 손실(loss)이 감소된 경우에 리프 노드 분리. 클수록 과적합 감소
5. `max_depth[default=6]` : 0 지정 시 깊이 제한 X. 높을수록 조건 특화된 룰 조건 생성 → 과적합 可, 보통 3~10 값 사용
6. `sub_sample[dafault=1]` : 트리가 커져 과적합되는 것을 방지하기 위해 데이터를 샘플링하는 비율 지정.

   일반적으로 0.5~1 값 사용

   (0.5 = 전체 50%를 트리 생성에 사용)
7. `colsample [default=1, alias: reg_lambda]` :트리 생성에 필요한 피처(칼럼)을 임의로 샘플링 (GBM의 max_features와 유사)

8. `lambda [default=1, alias: reg_lambda]` : L2 Regularization 적용 값. 피처 개수 ↑ → 적용 검토, 값 ↑ → 과적합 감소 효과
9. `alpha [default=0, alias: reg_alpha]` : L1 Regularization 적용 값. 위와 동일 효과
10. `scale_pos_weight [default=1]` : 치우친 비대칭 클래스로 구성된 데이터 세트의 균형 유지 위함

▼ 학습 테스크 파라미터

1. `objective` : 최솟값을 가질 손실 함수 정의.
2. `binary:logistic` : 이진 분류 시 적용
3. `multi:softmax` : 다중 분류 시 적용. 레이블 클래스 개수인 `num_class` 파라미터 지정 필요
4. `multi:softprob` : 개별 레이블 클래스의 해당되는 예측 확률 반환
5. `eval_metric` : 검증에 사용되는 함수 정의. (Default : 회귀 시 `rmse,` 분류 시 `error`)
    a. `rmse` - root mean square error
    b. `mae` - maen absolute error
    c. `logloss` - negative log-likelihood
    d. `merror` - binary classification error rate(0.5 threshold)
    e. `mlogloss` - multiclass logloss
    f. `auc` - area under the curve

- 과적합 문제 심각 시 다음 고려
    - `eta` 낮추기(0.01~0.1) & `num_round` / `n_estimators` 높이기
    - `max_depth` 낮추기
    - `min_child_weight` 높이기
    - `gamma` 값 높이기
    - `subsample`과 `colsample_bytree` 조정하기

XGBoost 특징

- 자체적으로 교차 검증, 성능 평가, 피처 중요도 등의 시각화 기능 가짐
- 조기 중단 기능 - 수행 시간 개선!
    - ex) n_estimators = 200, 조기 중단 파라미터 = 50

: 1~200회까지 부스팅 반복하다가, 50회를 반복하는 동안 학습 오류가 더 이상 감소하지 않으면 부스팅 종료

파이썬 래퍼 XGBoost 적용 - 위스콘신 유방암 예측

- 위스콘신 유방암 데이터 소개
  - 종양 크기, 모양 등 다양한 속성값 → 악성 종양(malignant) / 양성 종양(benign) 구분

1. xgboost 로딩, 사이킷런에 내장된 위스콘신 유방암 데이터 세트 Df로 로드

   ▼ 설명 부분

```
#buch 객체로 이루어진 데이터셋을 불러와 dataset 변수에 저장.
#이러면 다음에 매번 다시 지저분하게 안 불러와도 됨
#bunch 객체란?: data, target, feature_names 등 여러 속성 들어 있는 것
dataset = load_breast_cancer()

#data : dataset(아까 유방아마 bunch)의 features의 숫자 배열
#2차원 넘파이 배열 - (569, 30) -> 569명 환자 X 30개 특
features = dataset.data

#target : 정답 값(0 / 1)
labels = dataset.target
```

```
#data : df에 들어갈 실제 값. 여기선 569행 X 30열 표 들어갈 것
#columns : df의 열이름 지정. 리스트 or 배열 형태로 주어야 함
                feature_names는 이미 들어 있는 속성이라 사용하면 됨
cancer_df = pd.DataFrame(data=features, columns=dataset.feature
_names)

#결과인 'target' 열 추가, 위의 labels 넣음(원래 data에는 결과 없음)
cancer_df['target']=labels
```

2. 레이블 값 분포 확인
3. test/train set 분할
   - DMatrix
     - XGBoost만의 전용 데이터 객체. 이걸로 생성하여 모델에 입력해 줘야.

- Df, Series → DMatrix
- 파라미터 :
  1. `data` : 피처 세트
  2. `label` : 분류-레이블 세트/ 회귀-종속값 세트(숫자형)

4. XGBoost 하이퍼 파라미터 설정

▼ 설명 부분

```
params = {'max_depth':3,
        'eta':0.05, #학습률
        'objective':'binary:logistic', #0/1 이진 분류이므로
                                                목적함수는 이진로지스틱(bin
ary:logistic)
        'eval_metric':'logloss'     #오류함수의 평가 성능 지표
}
num_rounds=400  #부스팅 반복 횟수 400
```

5. 모델 학습
   - 평가용 데이터 세트 : 학습/평가용 데이터 세트 명기하는 개별 튜플 가진 리스트 형태로 설정
     - train set은 '`train`', test set은 '`eval`'로 명기

       ex) [(이름1, 'train'), (이름2, 'eval')]
6. test set 예측 수행
7. 모델 성능 평가
8. 시각화

사이킷런 래퍼 XGBoost의 개요 및 적용

- 사이킷런의 프레임워크와 연동하기 위해 사이킷런 전용 XGBoost 래퍼 클래스 개발
  - 사이킷런 기본 Estimator 그대로 상속 →fit(), predict()만으로 학습과 예측 가능
  - 기본 머신러닝 프로그램도 알고리즘 클래스만 XGBoost 래퍼 클래스로 바꾸면 사용 가능
  - 분류 래퍼 클래스 XGBClassifier, 회귀 래퍼 클래스 XGBRegressor
- 기존 xgboost 모듈에서 바뀐 파라미터

- eta → learning_rate
- sub_sample → subsample
- lambda → reg_lambda
- alpha → reg_alpha
- n_estimators = num_boost_round

4.7 LightGBM

개요

- XGBoost보다 시간 ↓, 메모리 사용량 ↓
- 기능상 다양성
- 리프 중심 트리 분할(Leaf Wise) 상용

  균형 트리 분할(Level Wise)

  : 최대한 균형 잡힌 트리 유지하면서 분할 → 트리 깊이 최소화

  - (+) : 오버피팅에 강함
  - (-) : 시간 ↑

  리프 중심 트리 분할(Leaf_wise)

  : 트리 균형 맞추지 X, 최대 손실 값(mex delta loss) 가지는 리프 노드를 지속적으로 분할 → 깊이 ↑, 비대칭적

  - (+) : 예측 오류 손실 최소화
- 카테고리형 피처의 자동 변할과 최적 분할(원핫인코딩 사용 필요 X)

LightBGM 하이퍼 파라미터

1. `num_iteration` [default =100] : 반복 수행하려는 트리 개수 지전
   a. 클수록 - 성능↑ / 과적합 성능↓
2. `learning_rate` [default = 0.1] : 0~1 사이 값으로 부스팅 스텝 시 학습률
3. `max_depth` [default =1] : `max_depth`과 동일
4. `min_data_in_leaf` [default =20] : min_samples_leaf와 동일
5. `num_leaves` [default =3] : 하나의 트리가 가질 수 있는 최대 리프 개수
6. `boosting` [default = gbdt] : 부스팅의 트리를 생성하는 알고리즘

a. gbdt - 일반적인 gradient boosting 결정 트리

b. rt : 랜덤 포레스트

7. `bagging_fraction` [default =1.0] : 데이터를 샘플링하는 비율 for 트리 커짐→과적합 방비
8. `feature_fraction` [default =1.0] : 개별 트리 학습 때마다 무작위 선택하는 피처 비율
9. `lambda_l2` [default =0.0] : L2 Regularization 제어 값
10. `lambda_l1` [default =0.0] : L1 Regularization 제어 값
11. `objective` : 최솟값을 가져야 할 손실함수 정의(회쉬, 다중 클래스 분류, 이진 분류 등)

하이퍼 파라미터 튜닝 방안

1. `num_leaves` 개수 중심으로, `min_child_samples(min_data_in_leaf)` & `max_depth` 함께 조정하며 복잡도 감소
2. `learning_rate` ↓ `n_estimators` ↑
3. `colsample_bytree` , `subsample` 로 train set에 사용할 피처 개수나 데이터 샘플링 레코드 개수 줄이기

파이런 래퍼 LightGBM과 사이킷런 XGBoost, LightGBM 하이퍼 파라미터 비교

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

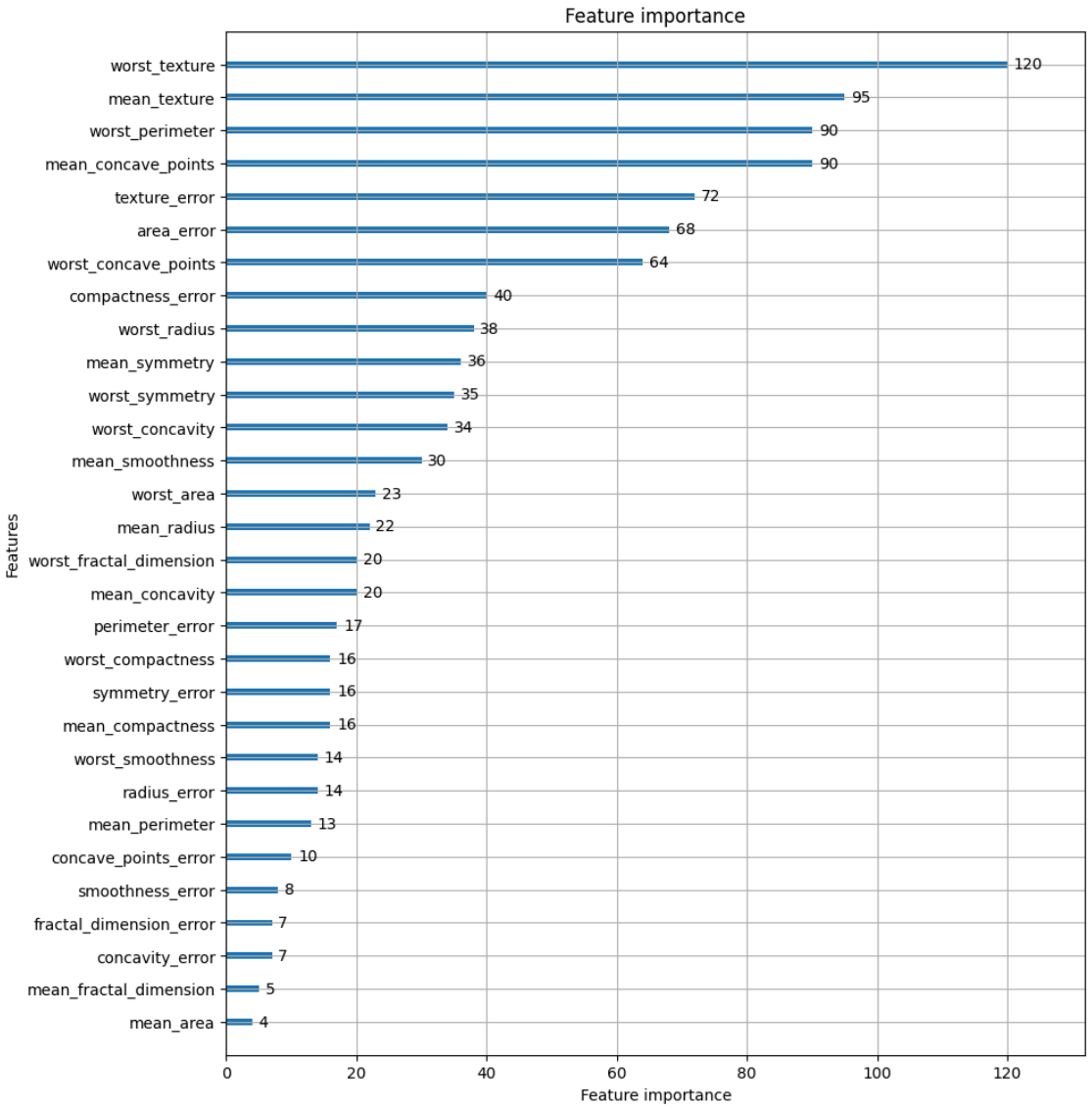
LightGBM 적용 - 위스콘신 유방암 예측

- 조기 중단 발생 확인 가능

- 예측 성능 :

```
오차 행렬
[[35  2]
 [ 2 75]]
정확도: 0.9649, 정밀도: 0.9740, 재현율: 0.9740, F1: 0.9740, AUC: 0.9877
```

- 피처 중요도



4.8 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

베이지안 최적화 개요

목적 함수 식을 제대로 알 수 없는 블랙 박스 형태의 함수에서, 최대/최소 함수 반환값을 만드는 최적 입력값을 가능한 적은 시도로 빠르고 효과적이게 찾아주는 방식

베이지안 확률에 기반을 두고 있는 최적화 방법

- 새 데이터 기반 → 사후 확률 개선
- 새 데이터 입력 → 최적 함수 예측 사후 모델 개선 → 최적 함수 모델 제작

중요 요소 : 대체 모델(Surrogate Model), 획득 함수(Acquisition Fuction)

- 대체 모델(Surrogate Model) : 획득 함수로부터 최적 함수를 예측할 수 있는 **입력값을 추천받은 뒤**, 최적 함수 모델 개선.
    - 획득 함수가 계산한 하이퍼 파라미터를 입력받으며 점차 개선

Step

1. 최초 랜덤 하이퍼 파라미터 샘플링 & 결과 관측
2. 관측 값 기반으로 대체 모델이 **최적 함수** 추정.
    - 예측 함수의 신뢰구간 포함(추정된 함수의 결괏값 오류 편차=함수의 불확실성)
    - 최적 관측값은 y축 value에서 가장 높은 값을 가질 때 하이퍼 파라미터
3. 추정된 최적 함수 기반으로 **획득 함수**가 **다음 관측할 하이퍼 파리미터** 값 계산.
    - 이전의 최전 관측값보다 더 큰 최댓값을 가질 가능성 높은 지점을 찾아 , 대체 모델에 전달
4. 획득 함수로부터 전달된 하이퍼파라미터 수행해 관측된 값 기반으로 **대체 모델 갱신**, 다시 최적 함수 예측 추정

HyperOpt 사용하기

- 주요 로직
    1. 입력 변수명과 입력값의 검색 공간 설정
    2. 목적 함수 설정
    3. 목적 함수의 반환 최솟값 가지는 최적 입력값 유추

1. 입력 변수명과 입력값의 검색 공간 설정
    - 입력 변수명, 입력값 검색 공간

    - 파이썬 딕셔너리 형태
    - 키값 : 입력 변수명
    - value값 : 해당 변수의 검색 공간
- hp 모듈 : 입력값의 검색 공간 다양 설정 가능하게 여러 함수 제공
    - `hq.quniform(label, low, high, q)` : 라벨로 지정된 입력값 변수 검색 공간을 low에서 high까지 q 간격으로 설정
    - `hp.uniform(label, low, hight)` : low에서 high까지 정규 분포 형태의 검색 공간 설정
    - `hp.randint(label, upper)` : 0~upper까지 랜덤 정숫값으로 검색 공간 설정
    - `hp.loguniform(label, low, hight)` : exp(uniform(low,hight)값 반환, 반환 값의 log 변환된 값은 정규 분포 형태를 가지는 검색 공간 설정
    - `hp.choice(label, options)` : 검색 값이 문자&숫자 섞여 있을 경우.
        - options : 리스트/튜플로 제공

2. 목적 함수 생성
    - 변숫값&검색 공간 가지는 딕셔너리를 인자로 받고, 특정 값을 반환하는 구조
    - 반환값은 숫자형 외에도 딕셔너리 형태로도 반환 可
3. 목적 함수의 반환 최솟값 가지는 최적 입력값 유추
    - 베이지안 최적화 기법에 기반하여 - `fmin()` 함수 제공
        - 주요 인자
            - `fn` : 위의 objective_func와 같은 목적 함수
            - `space` : 위의 search_space와 같은 검색 공간 딕셔너리
            - `algo` : 베이지안 최적화 적용 알고리즘
            - `max_evals` : 최적 입력값을 찾기 위한 입력값 시도 횟수
            - `trials` : 최적 입력값을 찾기 위해 시도한 입력값 & 해당 입력값의 목적 함수 반환값 결과 저장
            - `rstate` : random seed

HyperOpt를 이용한 XGBoost 하이퍼 파라미터 최적화

방법

1. 적용해야 할 하이퍼 파라미터와 검색 공간 설정
2. 목적 함수에서 XGBoost 학습 후 예측 성능 결과를 반환 값으로 설정
3. fmin() 함수에서, (목적 함수를 하이퍼 파라미터 검색 공간의 입력값들을 사용하여 최적의 예측 성능 결과를 반환하는) 최적 입력값들을 결정

4.10 분류 실습 - 캐글 신용카드 사기 검출

- 불균형한 레이블
  - 0 : 정상적인 신용카드 트랜잭션 데이터
  - 1 : 사기 트랜잭션(0.172%)

언더 샘플링과 오버 샘플링의 이해

- 이상 레이블 D 수 > 정상 레이블 D 수
  - 이상 레이블 D가 적어 제대로 다양한 유형 학습 X
  - ↔ 정상 레이블로 치우친 학습 수행
- 오버 샘플링
  - 적은 D 증식해 충분히 확보하는 방법
  - 동일X, 원본의 피처 값들을 약간만 변경해 증식
    - SMOTE : k-nn을 찾아, 이 D와 k개 이웃들의 차이 → 일정 값 → 기존과 약간 차이나는 새 D 생성
- 언더 샘플링
  - 많은 D를 적은 D 수준으로 감소

데이터 일차 가공 및 모델 학습/예측/평가

Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24
0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928
0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846
1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281

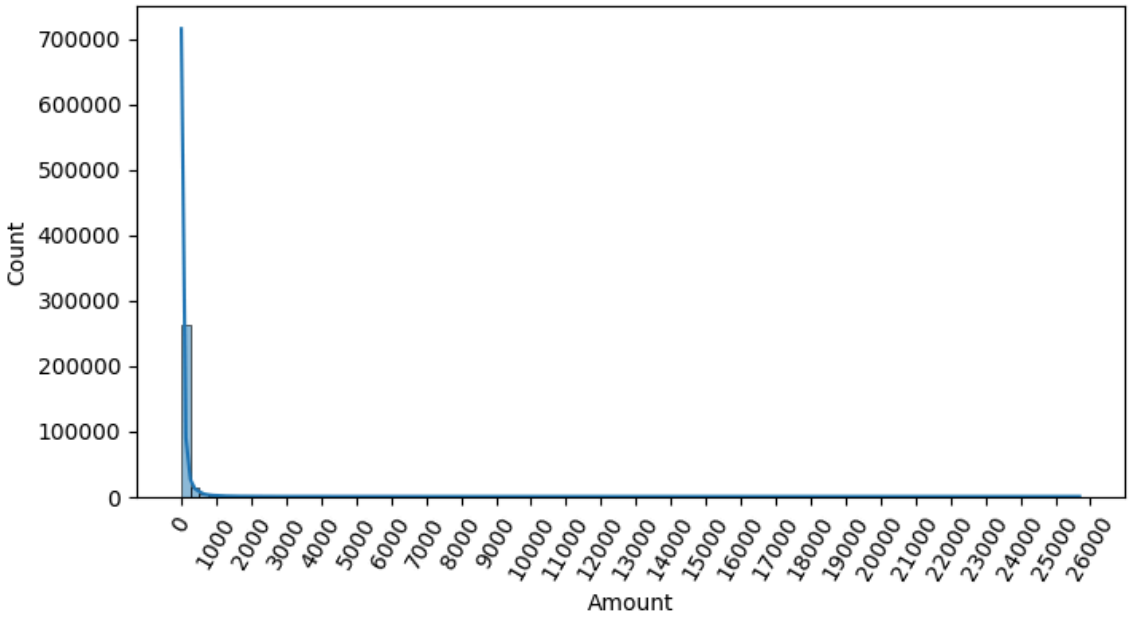
- V로 시작하는 피처들의 의미는 알 수 X
- Time 의미 X, 제거
- Amount : 금액

- 결과 :

```
[LightGBM] [Warning] Stopped training because there are no more leaves t
오차 행렬
[[33  4]
 [ 1 76]]
정확도: 0.9561, 정밀도: 0.9500, 재현율: 0.9870, F1: 0.9682, AUC: 0.9933
```

데이터 분포도 변환 후 모델 학습/예측/평가

- Amount 피처 분포도 확인



  - 1000불 이하인 데이터 대부분, 꼬리가 긴 분포 곡선.

→ 표준 정규 분포 형태로 변환 후 로지스틱 회귀 다시 성능 측정

```
### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85281    14]
 [   55    93]]
정확도: 0.9992, 정밀도: 0.8692, 재현율: 0.6284, F1: 0.7294, AUC: 0.9706
```

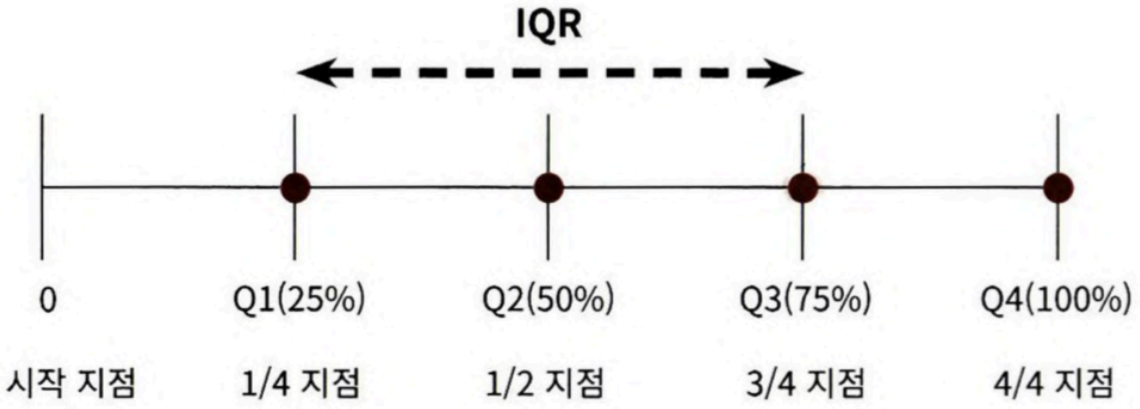
→ 로그 변환 후 다시 성능 측정

- 분포도 심하게 왜곡돼 있을때, 원래 값을 log으로 변환 → 원래 큰 값을 상대적으로 작은 값으로 변환 → 왜곡 개선
- 정밀도 향상, 재현율 저하

```
### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85282    13]
 [   59    89]]
정확도: 0.9992, 정밀도: 0.8725, 재현율: 0.6014, F1: 0.7120, AUC: 0.9734
```

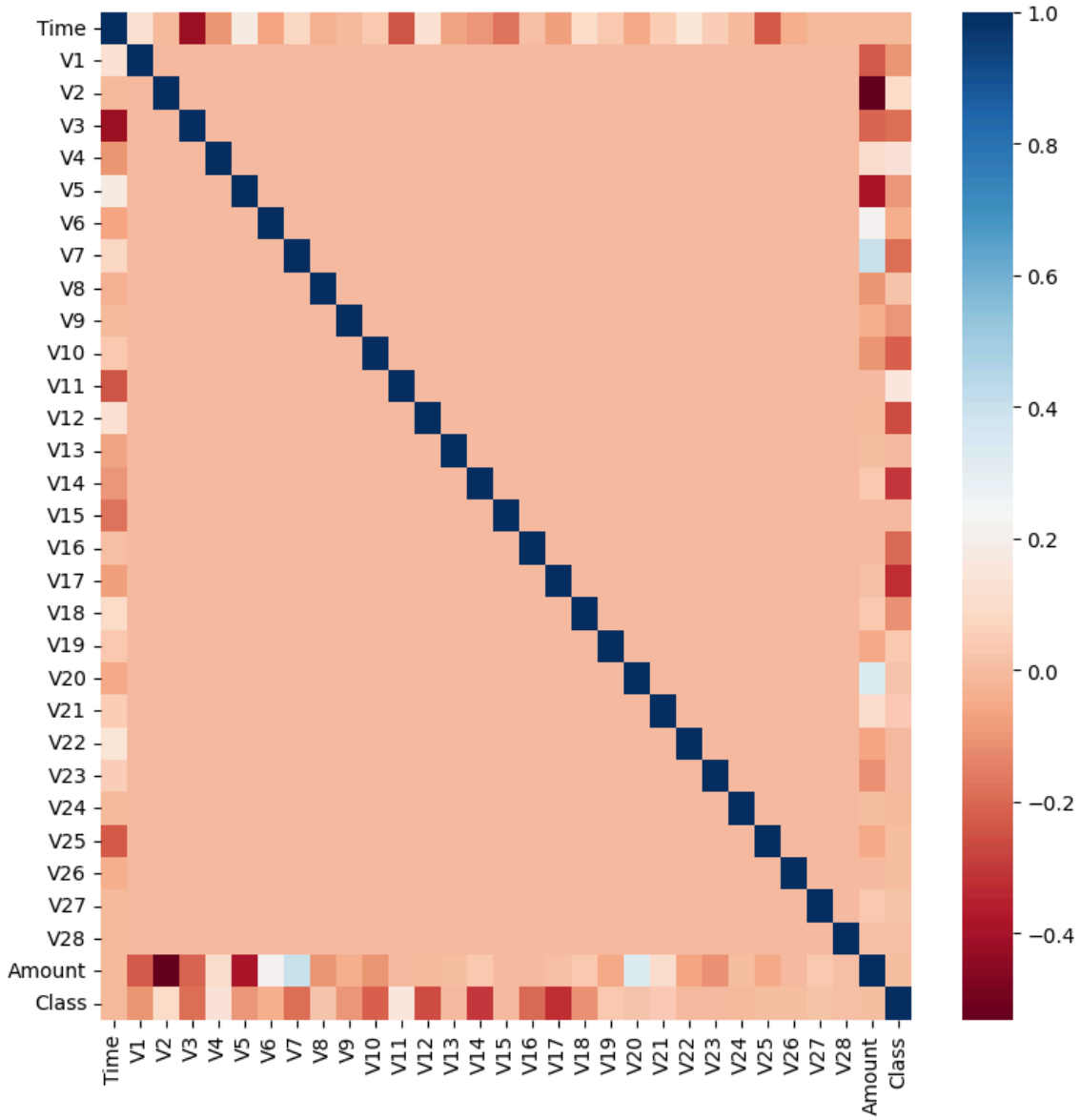
이상치 데이터 제거 후 모델 학습/예측/평가

- outlier : 전체 패턴에서 벗어난 이상 값 가진 D
- IQR : 사분위 값의 편차를 이용해 outlier 제거



  - Q1 ~ Q3 : IQR
    - IQR * 1.5로 생성된 범위 이용 → 최댓값/최솟값 결정 → 최댓값 초과, 최솟값 미달 D를 이상치로 간

- 피처별 상관도 구한 뒤 heatmap으로 시각화 : V14와 V17



- 결과 :

```
 ### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85280    15]
 [   48    98]]
정확도: 0.9993, 정밀도: 0.8673, 재현율: 0.6712, F1: 0.7568, AUC: 0.9725
```

SMOTE 오버 샘플링 적용 후 모델 학습/예측/평가

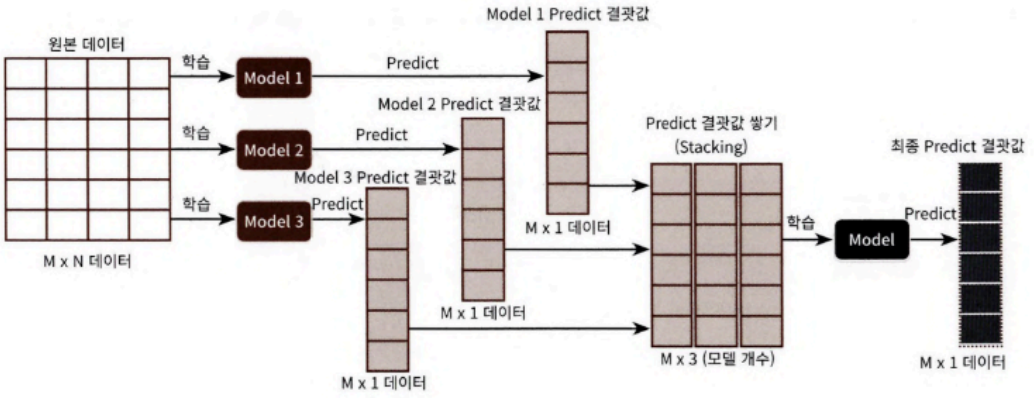
- 증식 후 성능 평가

```
오차 행렬
[[82933  2362]
 [   11   135]]
정확도: 0.9722, 정밀도: 0.0541, 재현율: 0.9247, F1: 0.1022, AUC: 0.9736
```

- 재현율 증가, 정밀도 급격 저하 ← 실제보다 너무 많은 Class=1을 학습해서 test set에서 예측 1 지나치게

4.11 스태킹 앙상블

- 예측값을 합하다 = 스태킹
- 개별 여러 알고리즘 결합 → 예측 결과 도출
- 배깅 / 부스팅과 차이점 : **개별 알고리즘으로 예측한 데이터를 기반으로, 다시 예측 수행**
  - 개별 알고리즘의 예측 결과 데이터 세트 → **최종 메타 데이터 세트** → 별도의 ML 알고리즘으로 **최종 학습** 수행 → 테스트 데이터 기반 다시 **최종 예측** 수행
  - 메타 모델 : 개별 모델의 예측된 데이터로 세트를 다시 기반으로 하여 학습하고 예측하는 방식
- 두 종류의 모델 필요 : 개별적인 기반 모델 + 최종 메타 모델(개별 기반 모델의 예측 데이터를 학습 데이터로 만들어서 학습)
- 여러 개별 모델의 예측 데이터 → 각각 스태킹 형태로 결합 → 최종 메타 모델의 학습용 피처 데이터 세트 & 테스트용 피처 데이터 세트 생성



기본 스태킹 모델

1. 스태킹에 사용될 ml 알고리즘 클래스 생성
   - 개별 모델 : KNN, RF, DT, ADABOOST

- 최종 모델 : Logistic Regression

2. 개별 모델 학습
3. 개별 모델의 예측 데이터 세트 반환, 개별 모델의 예측 정확도 확인

```
knn 정확도: 0.9211
rf 정확도: 0.9649
dt 정확도: 0.9123
ada 정확도: 0.9737
```

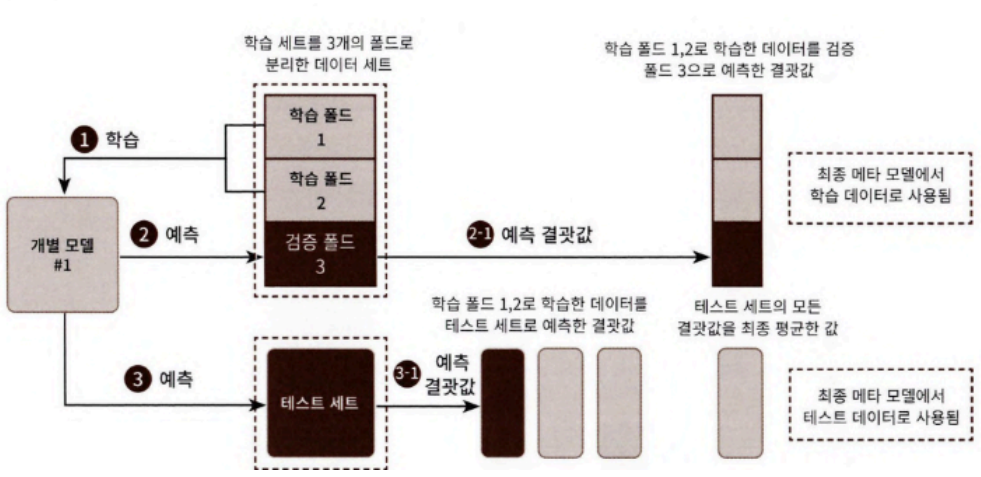
4. 개별로부터 얻은 예측값을 칼럼 레벨로 옆으로 붙여 피처 값으로 만들기
5. 최종 메타 모델에서 학습 데이터로 다시 사용

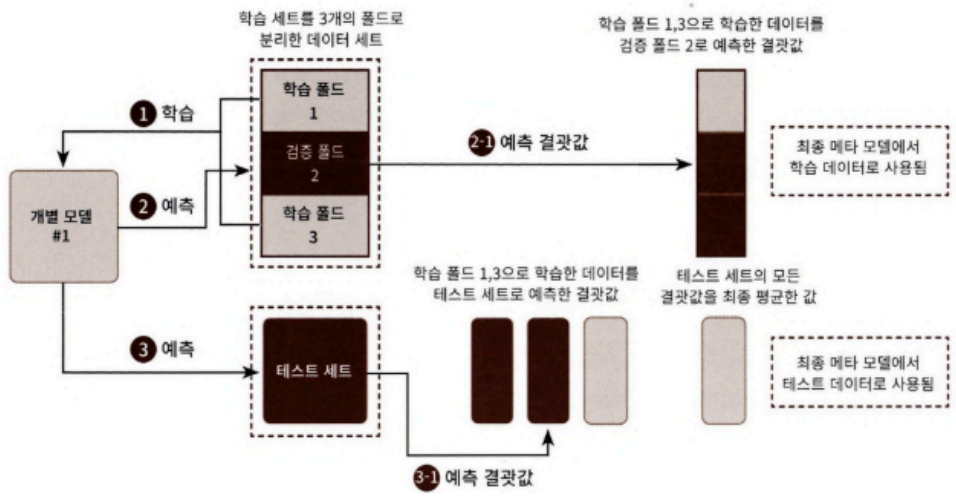
```
최종 메타 모델의 예측 정확도: 0.9737
```

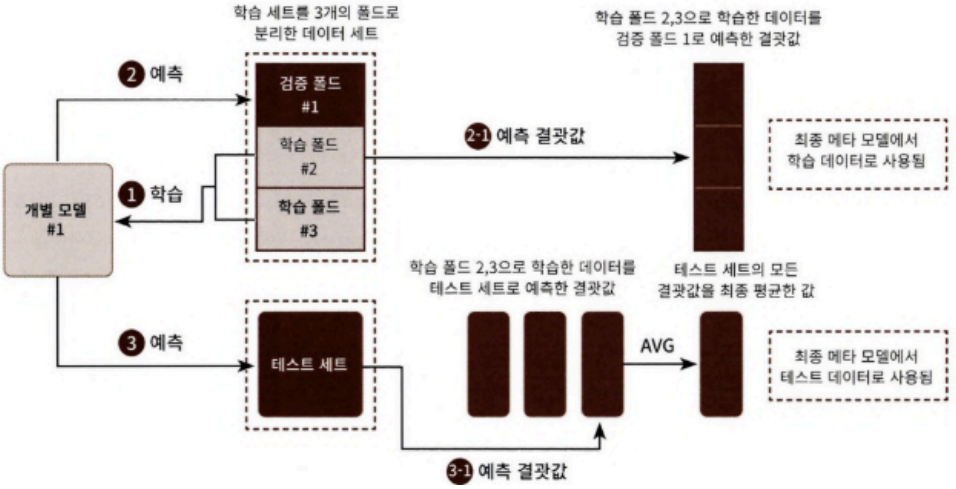
CV 세트 기반의 스태킹

- for 과적합 개선
- 최종 메타 모델을 위한 data set 만들 때, **교차 검증 기반으로 예측된** 결과 데이터 세트 이용

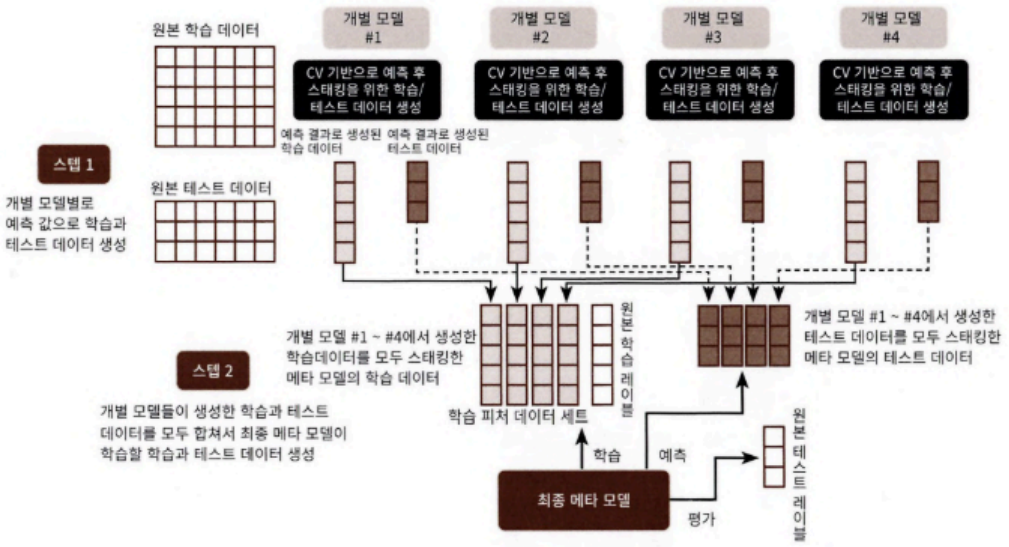
1. 각 모델별 예측 결과 값 기반 → 메타 모델 위한 train/test 데이터 생성







2. 1에서 개별 모델이 생성한 test/test 데이터를 모두 스태킹으로 합침 → 메타 모델이 최종 학습할 최종 trian/test 데이터 생성
3. **최종 생성 train** 데이터 + **원본 train** 데이터의 레이블 기반 학습 → **최종 생성** test 데이터 예측 → **원본** test 데이터의 레이블 데이터 기반 평가


원본 학습 데이터
개별 모델 #1
개별 모델 #2
개별 모델 #3
개별 모델 #4
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
CV 기반으로 예측 후 스태킹을 위한 학습/테스트 데이터 생성
예측 결과로 생성된 학습 데이터
예측 결과로 생성된 테스트 데이터
스텝 1
개별 모델별로 예측 값으로 학습과 테스트 데이터 생성
원본 테스트 데이터
개별 모델 #1 ~ #4에서 생성한 학습데이터를 모두 스태킹한 메타 모델의 학습 데이터
원본 학습 레이블
개별 모델 #1 ~ #4에서 생성한 테스트 데이터를 모두 스태킹한 메타 모델의 테스트 데이터
스텝 2
개별 모델들이 생성한 학습과 테스트 데이터를 모두 합쳐서 최종 메타 모델이 학습할 학습과 테스트 데이터 생성
학습 피처 데이터 세트
학습
예측
원본 테스트 레이블
최종 메타 모델
평가