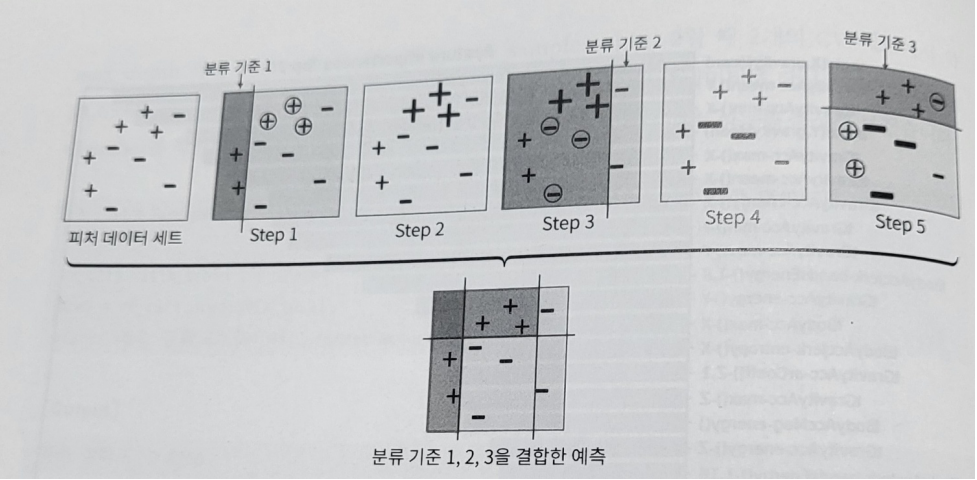
Euron 2주차 필사 (4.5~4.8, 4.11)

▼ 4.5 GBM(Gradient Boosting Machine)

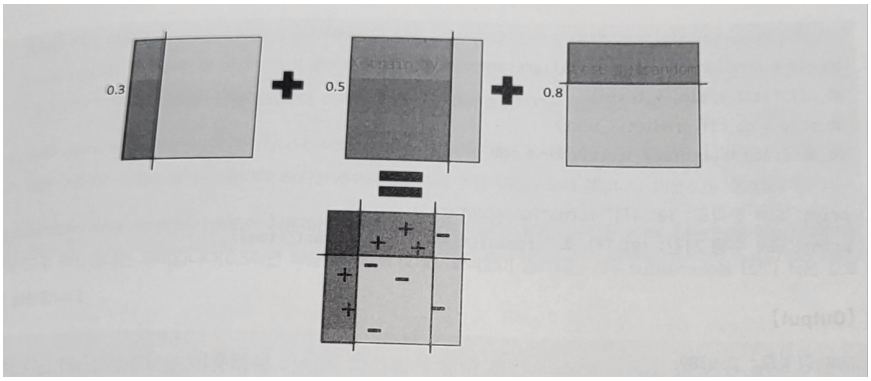
GBM의 개요 및 실습

- 부스팅 알고리즘 : 여러 개의 약한 학습기(weak nearner)를 순차적으로 학습-예측 → 잘못 예측한 데이터에 가중치를 부여 → 오류 개선해 나가면서 학습
  - 대표적인 구현 : AdaBoost(Adaptive boosting)와 그래디언트 부스트
    - AdaBoost(에이다 부스트) : 오류 데이터에 가중치를 부여하면서 부스팅 수행
    - 에이다 부스트 학습 방법



- Step 1 : 첫 번째 약한 학습기가 분류 기준 1로 +와 -를 분류한 것, 동그라미 안 +로 표시된 데이터 : + 데이터가 잘못 분류된 오류 데이터
- Step 2 : 오류 데이터에 대해 가중치 값 부여, 다음 약한 학습기가 더 잘 분류할 수 있게 + 크기가 커짐
- Step 3 : 두 번째 약한 학습기가 분류 기준 2로 +와 -를 분류, 동그라미 안 -로 표시된 데이터 : 잘못 분류된 오류 데이터
- Step 4 : 오류 데이터에 대해 가중치 값 부여, 다음 약한 학습기가 더 잘 분류할 수 있게 - 크기가 커짐
- Step 5 : 세 번째 약한 학습기가 분류 기준 3으로 +와 -를 분류

        - 마지막 : 1,2,3번째 약한 학습기를 모두 결합한 결과 예측, 개별 약한 학습기보다 정확도 훨씬 높아짐
    - ⇒ 에이다부스트는 약한 학습기가 순차적으로 오류 값에 대한 가중치를 부여한 예측 결정 기준을 모두 결합해 예측 수행



        - 위 그림과 같이 각각 가중치를 부여해 결합
            - 첫 번째 학습기에 가중치 0.3, 두 번째 학습기에 가중치 0.5, 세 번째 학습기에 가중치 0.8을 부여한 후 모두 결합해 예측 수행

- GBM(Gradient Boost Machine)
    - 에이다부스트와 유사
    - 가중치 업데이트를 경사 하강법(Gradient Descent)을 이용하는 것이 큰 차이
    - 과적합에도 강한 예측 성능을 가짐
    - 오류 값 = 실제 값 - 예측값
        - 분류의 실제 결과값 : y, 피처 : x1, x2, …, xn, 피처에 기반한 예측 함수 : F(x)
        - 오류식 : h(x) = y - F(x)
        - 경사 하강법
            - 오류식을 최소화하는 방향성을 가지고 반복적으로 가중치 값을 업데이트하는 것
            - 머신러닝에서 중요한 기법 중 하나
    - CART 기반 다른 알고리즘과 같이 분류, 회귀 가능
    - 사이킷런에서 GradientBoostingClassifier 클래스를 제공

- 사이킷런의 GBM을 이용해 사용자 행동 데이터 세트 예측 분류하기
  - 기본 하이퍼 파라미터만으로 93.89%의 예측 정확도로 랜덤 포레스트보다 나은 예측 성능을 나타냄
  - 일반적으로 예측 성능 : GBM > 랜덤 포레스트
- GBM의 단점
  - 수행 시간이 오래 걸림
  - 하이퍼 파라미터 튜닝 노력도 더 필요함
  - 사이킷런의 GradientBoostingClassifier는 약한 학습기의 순차적인 예측 오류 보정을 통해 학습 수행 → 멀티 CPU 코어 시스템을 사용하더라도 병렬 처리가 지원되지 않음 → 대용량 데이터의 경우 학습에 매우 많은 시간 소요
- **GBM 하이퍼 파라미터 소개**
  - loss : 경사 하강법에서 사용할 비용 함수를 지정, 기본값 'deviance'
  - learning_rate : GBM이 학습을 진행할 때마다 적용되는 학습률, Weak learner가 순차적으로 오류 값을 보정해 나가는 데 적용하는 계수
    - 0~1 사이 값 지정 가능, 기본값은 0.1
    - 너무 작은 값 적용? 업데이트 되는 값이 작아져 최소 오류 값을 찾아 예측 성능이 높아짐
    - 많은 weak learner는 순차적인 반복이 필요해 수행 시간이 오래 걸림
    - 너무 큰 값 적용? 최소 오류 값을 찾지 못하고 그냥 지나쳐 버려 예측 성능 저하됨 하지만 빠른 수행 가능
    - learning_rate는 n_estimators와 상호 보완하여 조합해 사용
      - learning_rate를 작게 하고 n_estimator를 크게 함
  - n_estimators : 약한 학습기의 개수 (개수가 많을수록 예측 성능이 일정 수준까지 좋아지지만 수행 시간 오래 걸림), 기본값은 100
  - subsample : 약한 학습기가 학습에 사용하는 데이터의 샘플링 비율, 기본값은 1
    - 과적합이 염려되는 경우 1보다 작은 값으로 설정

▼ 4.6 XGBoost(eXtra Gradient Boost)

- **XGBoost 개요**
  - 트리 기반 앙상블 학습에서 가장 각광받고 있는 알고리즘 중 하나
  - XGBoost의 장점
    - 뛰어난 예측 성능 : 일반적으로 분류와 회귀 영역에서 뛰어남
    - GBM 대비 빠른 수행 시간 : 병렬 수행 및 다양한 기능으로 빠른 성능 보장 (GBM 대비 빠르다는 것을 의미하지 다른 머신러닝 알고리즘에 비해 빠르다는 것은 아님)
    - 과적합 규제 : 자체에 과적합 규제 기능으로 과적합에 강한 내구성 가질 수 있음
    - Tree pruning(나무 가지치기) : 더 이상 긍정 이득이 없는 분할을 가지치기해서 분할 수를 줄임
    - 자체 내장된 교차 검증 : 반복 수행 시마다 내부적으로 학습 데이터 세트와 평가 데이터 세트에 대한 교차 검증을 수행해 최적화된 반복 수행 횟수를 가질 수 있음
      - 지정된 반복 횟수가 아니라 교차 검증을 통해 평가 데이터 세트의 평가 값이 최적화되면 반복을 중간에 멈출 수 있는 조기 중단 기능 있음
    - 결손값 자체 처리 : 결손값 자체 처리 기능 있음
  - XGBoost의 핵심 라이브러리 : C/C++로 작성됨
    - 파이썬 패키지(xgboost)도 제공
      - 초기에는 fit(), predict() 메서드와 같은 사이킷런 고유의 아키텍처가 적용될 수 없었음 → 사이킷런과 연동할 수 있는 Wrapper class 제공
    - 사이킷런 래퍼 클래스 : XGBClassifier, XGBRegressor
      - 사이킷런 estimator가 학습을 위해 사용하는 fit()과 predict()와 같은 표준 사이킷런 개발 프로세스 및 다양한 유틸리티 활용
- **파이썬 래퍼 XGBoost 하이퍼 파라미터**
  - GBM과 유사한 하이퍼 파라미터를 동일하게 가짐
  - 조기 중단, 과적합을 규제학 위한 하이퍼 파라미터 등이 추가됨
  - 파이썬 래퍼 XGBoost 하이퍼 파라미터

- 일반 파라미터 : 실행 시 스레드의 개수나 slient 모드 등의 선택을 위한 파라미터, 디폴트 파라미터 값을 바꾸는 경우는 거의 없음
    - 주요 일반 파라미터
        - booster : gbtree 또는 gblinear 선택, 디폴트는 gbtree
        - silent : 디폴트는 0, 출력 메시지를 나타내고 싶지 않은 경우 1로 설정
        - nthread : CPU의 실행 스레드 개수를 조정, 디폴트는 전체 스레드를 다 사용
            - 멀티 코어/스레드 CPU 시스템에서 전체 CPU를 사용하지 않고 일부 CPU만 사용해 ML 어플리케이션을 구동하는 경우에 변경
- 부스터 파라미터 : 트리 최적화, 부스팅, regularization 등과 관련된 파라미터
    - 주요 부스터 파라미터
        - eta : GBM의 학습률과 같은 파라미터, 0~1 사이의 값을 지정하며 부스팅 스텝을 반복적으로 수행할 때 업데이트되는 학습률 값
            - 파이썬 래퍼 기반 xgboost : 디폴트 0.3
            - 사이킷런 래퍼 클래스 이용 : learning_rate 파라미터로 대체, 디폴트는 0.1
            - 보통 0.01~0.2 사이의 값을 선호
        - num_boost_rounds : GBM의 n_estimators와 같은 파라미터
        - min_child_weight : 트리에서 추가적으로 가지를 나눌지를 결정하기 위해 필요한 데이터들의 weight 총합. 클수록 분할을 자제하며 과적합 조절에 사용됨
            - 디폴트 : 1
        - gamma(디폴트=0) : 트리의 리프 노드를 추가적으로 나눌지를 결정할 최소 손실 감소 값
            - 해당 값보다 큰 손실이 감소된 경우에 리프 노드를 분리
            - 값이 클수록 과적합 감소 효과가 있음

        - max_depth(디폴트=6) : 트리 기반 알고리즘의 max_depth와 같음
            - 0으로 지정하면 깊이에 제한이 없음
            - 높은 값이면 과적합 가능성이 높아짐, 보통 3~10 사이 값을 적용
        - sub_sample(디폴트=1)
            - GBM의 subsample과 동일
            - 과적합 제어, 0.5로 지정하면 전체 데이터의 절반을 트리를 생성하는 데 사용
            - 일반적으로 0.5~1 사이의 값을 사용
        - colsample_bytree(디폴트=1) : GBM의 max_features와 유사
            - 트리 생성에 필요한 피처를 임의로 샘플링하는 데 사용됨, 과적합 조정
        - lambda(디폴트=1) : L2 Regularization 적용 값
            - 피처 개수가 많을 경우 적용을 검토, 값이 클수록 과적합 감소 효과가 있음
        - alpha(디폴트=0) : L1 Regularization 적용 값
            - 피처 개수가 많을 경우 적용을 검토, 값이 클수록 과적합 감소 효과가 있음
        - scale_pos_weight(디폴트=1) : 특정 값으로 치우친 비대칭한 클래스로 구성된 데이터 세트의 균형을 유지하기 위한 파라미터
    - 학습 태스크 파라미터 : 학습 수행 시의 객체 함수, 평가를 위한 지표 등을 설정하는 파라미터
        - objetive : 최솟값을 가져야 할 손실 함수를 정의
        - binary:logistic : 이진 분류일 때 적용
        - multi:softmax : 다중 분류일 때 적용, 손실함수가 다중 분류일 때 레이블 클래스의 개수인 num_class 파라미터를 지정해야 함
        - multi:softprob : multi:softmax와 유사, 개별 레이블 클래스의 해당되는 예측 확률을 반환

        - eval_metric : 검증에 사용되는 함수 정의, 기본값은 회귀인 경우 rmse, 분류일 경우 error
            - rmse, mae, logloss, error, merror, mlogloss, auc가 있음
    - ⇒ 뛰어난 알고리즘일수록 파라미터 튜닝 필요성이 적음
        - 파라미터 튜닝에 들이는 노력 대비 성능 향상 효과가 높지 않은 경우가 대부분
    - 과적합 문제가 심각하다면?
        - eta 값을 낮춘다. num_round는 반대로 높여줘야 한다.
        - max_depth 값을 낮춘다.
        - min_child_weight 값을 높인다.
        - gamma 값을 높인다.
        - subsample과 colsample_bytree를 조정
    - 조기 중단 기능
        - 수행 속도를 향상시키기 위한 대표적인 기능
        - 기본 GBM에는 조기 중단 기능 없음
            - n_estimators에 지정된 횟수만큼 반복적으로 학습 오류를 감소시키며 학습을 진행, 중간에 반복을 멈출 수 없고 지정된 횟수를 다 완료해야 함
        - XGBoost, LightGBM은 모두 조기 중단 기능이 있음
            - n_estimators에 지정한 부스팅 반복 횟수에 도달하지 않더라도 예측 오류가 더 이상 개선되지 않으면 반복을 끝까지 수행하지 않고 중지해 수행 시간을 개선할 수 있음
            - 예) n_estimators=200, 조기 중단 파라미터 값=50 → 1~200회까지 부스팅 반복하다가 50회를 반복하는 동안 학습 오류가 감소하지 않으면 더 이상 부스팅을 진행하지 않고 종료
                - 100회에서 학습 오류 값이 0.8인데 101~150회 반복하는 동안 예측 오류가 0.8보다 작은 값이 하나도 없으면 부스팅 종료

- **파이썬 래퍼 XGBoost 적용 - 위스콘신 유방암 예측**
    - 위스콘신 유방암 데이터 활용, 파이썬 래퍼 XGBoost API 사용법 알아보기

- xgboost는 자체적으로 교차 검증, 성능 평가, 피처 중요도 등의 시각화 기능을 가짐
- 조기 중단 기능이 있음
- 위스콘신 유방암 데이터 세트
  - 종양의 크기, 모양 등의 다양한 속성값을 기반으로 악성 종양인지 양성 종양인지를 분류한 데이터 세트
  - 종양 : 양성 종양과 악성 종양(빠르게 성장, 확산 및 전이로 인해 생명 위협)으로 구분됨
- XGBoost를 이용해 종양의 다양한 피처에 따라 악성종양인지 양성종양인지 예측
  - plot_importance : 피처의 중요도 시각화 모듈
  - load_breast_cancer() 호출
  - 타깃 레이블 값 종류 : 악성='malignant'=0 / 양성='benign'=1
  - 레이블 값의 분포 : 양성 357개, 음성 212개
  - 데이터 세트의 80%를 학습용, 20%를 테스트용으로 추출 → 80%의 학습용 데이터에서 90%가 최종 학습용, 10%가 검증용
    - 검증용 데이터 세트를 별도로 분할하는 이유 : 검증 성능 평가와 조기 중단 수행
  - 전체 569개의 데이터 세트에서 최종 학습용 409개, 검증용 46개, 테스트용 114개가 추출됨
  - train()으로 학습을 수행하면서 반복 시마다 train-logloss와 eval-logloss가 지
  - num_boost_round를 500회로 설정했음에도 불구하고 학습은 176번째 반복에서 완료됨 → 126번째 반복에서 logloss값이 가장 낮음, 126~76번까지 early_stopping_rounds로 지정된 50회 동안 logloss 값은 이보다 향상되지 않음 → 반복 멈춤
  - predict()로 예측, 예측 확률이 0.5보다 크면 1, 그렇지 않으면 0으로 예측 값 결정
  - get_clf_eval() 함수를 적용해 모델 예측 성능 평가
  - plot_importance() API : 피처의 중요도를 막대그래프 형식으로 나타냄 (f스코어 기반)

            - f스코어 : 해당 피처가 트리 분할 시 얼마나 자주 사용되었는지를 지표로 나타낸 값
            - 호출 시 파라미터로 앞에서 학습이 완료된 모델 객체 및 맷플롯립의 ax 객체를 입력하기만 하면 됨
            - 넘파이 기반 피처 데이터로 학습 시 f0, f1과 같이 피처 순서별로 f 자 뒤에 순서를 붙여 피처명 나타냄
        - to_graphviz() API : 규칙 트리 구조 시각화
        - cv() API : 교차 검증 수행 후 최적 파라미터 구할 수 있는 방법
            - xgb.cv의 반환값 : DataFrame 형태
- 파이썬 래퍼 XGBoost의 특징
    - XGBoost만의 전용 데이터 객체인 Matrix 사용 → Lumpy 또는 Pandas로 되어 있는 학습용, 검증, 테스트용 데이터 세트를 모두 전용의 데이터 객체인 DMatrix로 생성하여 모델을 입력해줘야 함
    - 현 버전은 넘파이 외에도 DataFrame과 Series 기반으로도 DMatrix를 생성할 수 있음
    - DMatrix의 파라미터 : data(피처 데이터 세트), label(분류 : 레이블 데이터 세트 / 회귀 : 숫자형 종속값 데이터 세트)
        - 넘파이, DataFrame, Series, libsvm txt 포맷 파일, xgboost 이진 버퍼 파일을 파라미터로 입력하여 변환할 수 있음
    - 과거 버전에서 판다스의 DataFrame과 호환되지 않아서 DMatrix 생성 시 오류가 발생한 경우 : DataFrame.values를 이용해 넘파이로 일차 변환 → DMatrix 변환 적용
    - xgboost를 이용해 학습하기 전 먼저 XGBoost의 하이퍼 파라미터를 설정
        - 주로 딕셔너리 형태로 입력
    - 지정된 하이퍼 파라미터로 모델 학습 : 하이퍼 파라미터를 train() 함수에 파라미터로 전달
    - 학습 시 수행 속도 개선을 위해 조기 중단 기능 제공
        - 조기 중단의 성능 평가 : 별도의 검증 데이터 세트 이용
        - xgboost의 train() 함수에 early_stopping_rounds 파라미터를 입력하여 설정함

            - 반드시 평가용 데이터 세트 지정과 eval_metric을 함께 설정해야 함
                - 분류일 경우 주로 error, logloss를 적용
            - eval_metric의 지정된 평가 지표로 예측 오류 측정
            - 평가용 데이터 세트는 학습과 평가용 데이터 세트를 명기하는 개별 튜플을 가지는 리스트 형태로 설정
                - 예) [(dtr, 'train'), (dval, 'eval')]
    - train()은 학습이 완료된 모델 객체 반환
    - predict() 메서드
        - 예측을 위함, 예측 결과값이 아닌 예측 결과를 추정할 수 있는 확률 값을 반환함
- **사이킷런 래퍼 XGBoost의 개요 및 적용**
    - 사이킷런 기본 Estimator를 그대로 상속 → 사이킷런의 다른 유틸리티를 그대로 사용 가능
    - 기존 다른 머신러닝 알고리즘으로 만들어놓은 프로그램이 있어도 알고리즘 클래스만 XGboost 래퍼 클래스로 바꾸면 기존 프로그램 그대로 사용 가능
    - 분류를 위한 래퍼 클래스(XGBClassifier), 회귀를 위한 래퍼 클래스(XGBRegressor)
    - 하이퍼 파라미터 : learning_rate, subsample, reg_lambda, reg_alpha 등
        - n_estimators와 num_boost_round는 동일 → 둘 다 쓰이면 n_estimators 적용
    - 위스콘신 대학병원 유방암 데이터 세트 예측 - XGBClassifier 이용
        - 파이썬 래퍼 XGBoost보다 더 좋은 평가 결과가 나옴 → 위스콘신 데이터 세트 개수가 작기 때문에 전반적으로 검증 데이터를 분리하거나 교차 검증등을 적용할 때 성능 수치가 불안정하기 때문
        - 데이터 건수가 많으면 일반적으로는 과적합 개선 가능 → 모델 성능 향상 가능
        - 파이썬 래퍼와 동일하게 176번째 반복에서 학습 마무리
    - 조기 중단 : 파라미터를 fit()에 입력하면 됨
        - 파라미터 : early_stopping_rounds, eval_metric, eval_set

        - early_stopping_rounds를 너무 급격하게 줄이면 성능 향상 여지 있음에도 불구하고 반복 멈춤 → 성능 나빠질 가능성 있음

▼ 4.7 LightGBM

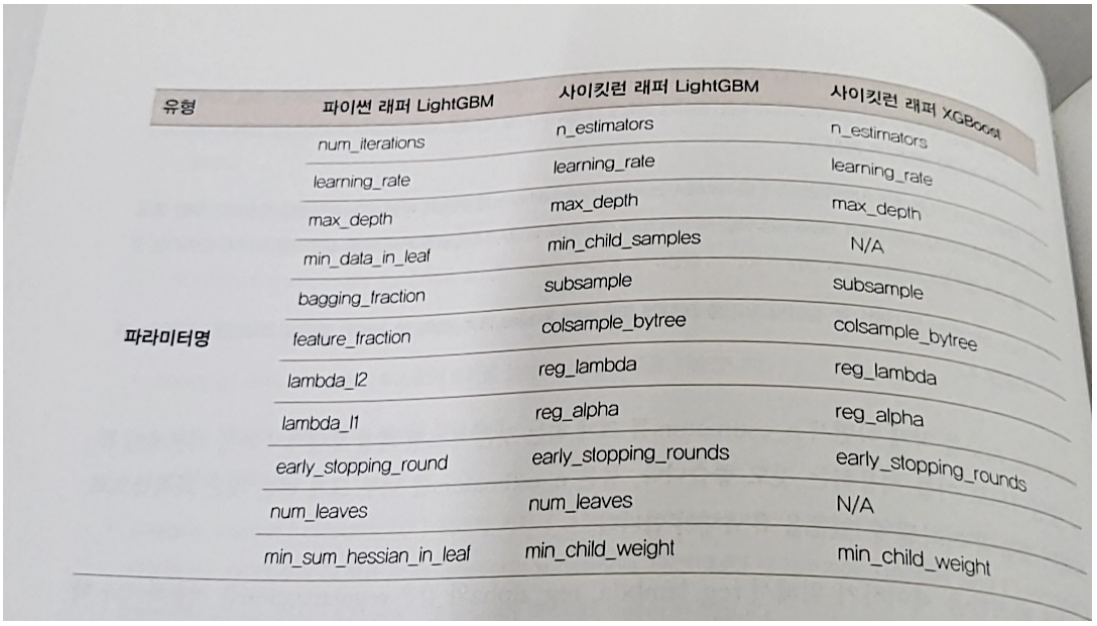
- 부스팅 계열 알고리즘에서 각광을 받고 있음
- 장점
    - XGBoost보다 학습에 걸리는 시간이 훨씬 적음
    - 메모리 사용량도 상대적으로 적음 → (’Light’GBM)
    - 카테고리형 피처의 자동 변환과 최적 분할(원-핫 인코딩 등을 사용하지 않고도 카테고리형 피처를 최적으로 변환하고 이에 따른 노드 분할 수행)
- 단점
    - 적은 데이터 세트(10,000건 이하 데이터 세트 정도)에 적용할 경우 과적합이 발생하기 쉬움
- 특징
    - 예측 성능은 XGBoost와 별다른 차이 없음
    - 기능상 다양성은 LightGBM > XGBoost (XGBoost보다 2년 후에 만들어져 장점 계승, 단점 보완)
    - 리프 중심 트리 분할 방식(Leaf Wise)을 사용
        - 트리 균형을 맞추지 않고, 최대 손실 값을 가지는 리프 노드를 지속적으로 분할 → 트리 깊이가 깊어지고 비대칭적인 규칙 트리 생성 → 학습 반복할수록 예측 오류 손실을 최소화함
        - 기존 대부분 트리 기반 알고리즘은 균형 트리 분할 방식을 사용 → 트리 깊이 최소화, (+) 과적합에 강한 구조, (-) 균형을 맞추기 위한 시간 소요
    - 파이썬 패키지명 : ‘lightgbm’ → 초기에는 파이썬 래퍼용만 개발되었고 사이킷런 래퍼가 추가로 개발됨
        - 사이킷런 래퍼 LightGBM 클래스 : 분류(LGBMClassfier), 회귀(LGBMRegressor)
- **LightGBM 하이퍼 파라미터**
    - XGBoost와 많은 부분 유사
    - XGBoost와 다르게 리프 노드가 계속 분할되면서 트리의 깊이가 깊어지므로 트리 특성에 맞는 하이퍼 파라미터 설정이 필요함 (예: max_depth를

매우 크게 가짐)

- 주요 파라미터
  - num_iterations (디폴트=100) : 반복 수행하려는 트리의 개수 지정
    - 크게 지정할수록 예측 성능 높아짐, 너무 크게 지정하면 과적합으로 성능 저하 가능성 있음
    - 사이킷런 호환 클래스에서는 n_estimators로 이름 변경됨
  - learning_rate(디폴트=0.1) : GBM, XGBoost와 동일한 파라미터
  - max_depth(디폴트=-1) : 트리 기반 알고리즘과 동일한 파라미터
    - 0보다 작은 값을 지정하면 깊이에 제한 없음. LightGBM은 다른 트리와 다르게 이 값이 더 큰 편
  - min_data_in_leaf (디폴트=20) : 결정 트리의 min_samples_leaf와 같은 파라미터
    - LightGBMClassifier에서는 min_child_samples 파라미터로 이름 변경됨
  - num_leaves(디폴트=31) : 하나의 트리가 가질 수 있는 최대 리프 개수
  - boosting(디폴트=gbdt) : 부스팅 트리 생성 알고리즘 기술
  - bagging_fraction(디폴트=1.0) : 과적합 방지용 데이터 샘플링 비율 지정, subsample과 동일
  - feature_fraction(디폴트=1.0) : 개별 트리를 학습할 때마다 무작위로 선택하는 피처의 비율, 과적합 방지용, max_features와 유사
  - lambda_l2(디폴트=0.0) : L2 regulation 제어를 위한 값
    - 피처 개수가 많을 경우 적용 검토, 값이 클수록 과적합 감소 효과 있음
  - lambda_l2(디폴트=0.0) : L2 regulation 제어를 위한 값
    - 피처 개수가 많을 경우 적용 검토, 값이 클수록 과적합 감소 효과 있음
  - objective : 최솟값을 가져야 할 손실함수를 정의

- **하이퍼 파라미터 튜닝 방법**

    - num_leaves 개수를 중심으로 min_child_samples, max_depth를 함께 조정 → 모델 복잡도 줄임
    - learning_rate를 작게 하면서 n_estimators를 크게 함 (너무 크게 하면 과적합 가능성)
    - 이외에도 reg_lambda, reg_alpha, colsample_bytree, subsample 파라미터 적용 가능
- 파이썬 래퍼 LightGBM 하이퍼 파라미터 vs 사이킷런 래퍼 XGBoost, LightGBM 하이퍼 파라미터



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**
    - lightgbm에서 LGBMClassifier를 임포트해 사용
    - 조기 중단 가능 → fit()에 조기 중단 관련 파라미터 설정하면 됨
        - 조기 중단으로 11번 반복까지만 수행 후 학습 종료
    - plot_importance() 제공

▼ 4.8 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

- Grid Search 방식
    - 하이퍼 파라미터 튜닝을 위해 사이킷런에서 제공함.
    - 튜닝해야 할 하이퍼 파라미터 개수가 많을 경우 최적화 수행 시간이 오래 걸림

    - 개별 하이퍼 파라미터 값의 범위가 넓거나 학습 데이터가 대용량인 경우 최적화 시간이 더욱 늘어남
    - XGBoost, LightGBM : 하이퍼 파라미터 개수가 다른 알고리즘 대비 많음 → Grid Search 방식은 많은 시간이 소모됨
        - 어쩔 수 없이 하이퍼 파라미터 개수를 줄이거나 개별 하이퍼 파라미터 범위를 줄여야 함
    - ⇒ 실무의 대용량 학습 데이터에 XGBoost나 LightGBM의 하이퍼 파라미터 튜닝 시 : 베이지안 최적화 기법 사용
- 베이지안 최적화 기법
    - 목적 함수 식을 제대로 알 수 없는 블랙 박스 형태의 함수에서 최대 또는 최소 함수 반환 값을 만드는 최적 입력값을 가능한 적은 시도를 통해 빠르고 효과적으로 찾아주는 방식
    - 예) 특정 함수 식을 알고 있다 → 쉽게 반환값을 최대/최소로 하는 값을 찾을 수 있음
        - 함수 식 자체를 알 수 없고, 단지 입력값과 반환값만 알 수 있는 상황이라면? 함수 반환값의 최대/최소 값을 찾기란 매우 어려움
        - 함수 식 자체가 복잡하고, 입력값의 개수가 많거나 범위가 넓은 경우 빠르게 최적 입력값 찾기 어려움 ⇒ 베이지안 최적화 이용하면 쉽게 찾을 수 있음
    - 베이지안 확률에 기반을 두고 있음
        - 새로운 데이터를 입력받았을 때 최적 함수를 예측하는 사후 모델을 개선해나가면서 최적 함수 모델 만들어 냄
    - 두 가지 중요 구성 요소
        - 대체 모델(Surrogate Model)
            - 획득 함수로부터 최적 함수를 예측할 수 있는 입력값을 추천 받음 → 최적 함수 모델 개선
            - 획득 함수가 계산한 하이퍼 파라미터를 입력받으면서 점차적으로 개선됨
            - 일반적으로 가우시안 프로세스 적용
                - HyperOpt는 트리 파르젠 Estimator(TPE)를 사용
        - 획득 함수(Acquisition Function)

        - 개선된 대체 모델을 기반으로 최적 입력값 계산
        - 개선된 대체 모델을 기반으로 획득 함수는 더 정확한 하이퍼 파라미터를 계산할 수 있게 됨
    - 베이지안 최적화 단계
        1. 최초에는 랜덤하게 하이퍼 파라미터들을 샘플링, 성능 결과 관측
        2. 관측된 값을 기반으로 대체 모델은 최적 함수를 추정
        3. 추정된 최적 함수를 기반으로 획득 함수는 다음으로 관측할 하이퍼 파라미터 값을 계산, 이전의 최적 관측값보다 더 큰 최댓값을 가질 가능성이 높은 지점을 찾아서 다음에 관측할 하이퍼 파라미터를 대체 모델에 전달
        4. 획득 함수로부터 전달된 하이퍼 파라미터를 수행 → 관측된 값을 기반으로 대체 모델은 갱신 → 다시 최적 함수 예측, 추정
        - 3단계와 4단계를 특정 횟수만큼 반복하면 대체 모델의 불확실성 개선, 정확한 최적 함수 추정이 가능해짐
- **HyperOpt 사용하기**
    - 베이지안 최적화를 머신러닝 모델의 하이퍼 파라미터 튜닝에 적용할 수 있게 제공하는 파이썬 패키지 중 하나
    - 주요 로직
        1. 입력 변수명과 입력값의 검색 공간 설정
            a. hp 모듈 이용
            b. 파이썬 딕셔너리 형태로 설정되어야 함 (키 값 : 입력 변수명, 밸류 값 : 검색 공간)
            c. 검색 공간을 제공하는 대표적인 함수
                i. hp.quniform(label, low, hight, q) : label로 지정된 입력값 변수 검색 공간을 최솟값 low에서 최댓값 high까지 q의 간격을 가지고 설정
                ii. hp.uniform(label, low, high) : 최솟값에서 최댓값까지 정규 분포 형태의 검색 공간 설정
                iii. hp.randint(label, upper) : 0부터 최댓값 upper까지 random한 정숫값으로 검색 공간 설정

iv. hp.loguniform(label, low, high) : exp.uniform(low, high)값을 반환, 반환 값의 log 변환된 값은 정규 분포 형태를 가지는 검색 공간 설정

v. hp.choice(label, options) : 검색 값이 문자열 또는 문자열과 숫자값과 섞여 있을 경우 설정

2. 목적 함수 설정
   a. 반드시 딕셔너리를 인자로 받고, 특정 값을 반환하는 구조로 만들어져야 함
   b. 반환값은 숫자형 단일값 외에도 딕셔너리 형태로 반환 가능
      i. 딕셔너리 형태로 반환할 경우 loss와 status 키 값을 설정해서 반환해야 함
3. 목적 함수의 반환 최솟값을 가지는 최적 입력값 유추
   a. fmin(objective, space, algo, max_evals, trials) 함수 제공
      i. fn : 위에서 생성한 목적 함수
      ii. space : 위에서 생성한 검색 공간 딕셔너리
      iii. algo : 베이지안 최적화 적용 알고리즘(기본적으로 TPE)
      iv. max_evals : 최적 입력값을 찾기 위한 입력값 시도 횟수
      v. trials : 최적 입력값을 찾기 위해 시도한 입력값 및 해당 입력값의 목적 함수 반환값 결과를 저장하는 데 사용됨. Trials 클래스를 객체로 생성한 변수명을 입력
      vi. rstate : fmin()을 수행할 때마다 동일한 결과값을 가질 수 있도록 설정하는 랜덤 시드 값

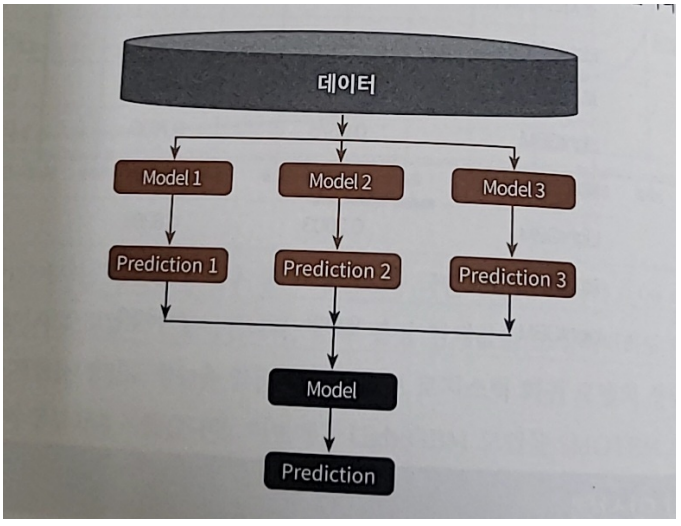
- 베이지안 최적화 수행
  - fmin() 함수 호출
    - max_evals=5, fn 인자는 objective_func, 검색 공간은 search_space, algo는 기본값으로 설정
    - 일반적으로 rstate는 잘 적용하지 않음. 일반적인 정수형 값을 넣지 않음
    - 20회 반복 시 목적 함수의 최적 최솟값을 근사할 수 있는 결과 도출됨 → 상대적으로 최적 값 찾는 시간을 많이 줄여 줌
  - Trials 객체

        - fmin() 함수 수행 시 인자로 들어감
        - 함수의 반복 수행 시마다 입력되는 변숫값들과 함수 반환값을 속성으로 가짐
        - 중요 속성
            - results
                - 함수 반복 수행 시마다 반환되는 반환값, 리스트 형태
                - 리스트 내 개별 원소는 {'loss': 함수 반환값, 'status':반환 상태값} 같은 딕셔너리
            - vals
                - 함수 반복 수행 시마다 입력되는 입력 변숫값
                - 딕셔너리 형태로 값을 가짐
                - 입력값들은 리스트 형태로 가짐
        - results와 vals 속성값들을 DataFrame으로 만들 수 있음
- **HyperOpt를 이용한 XGBoost 하이퍼 파라미터 최적화**
    1. 적용해야 할 하이퍼 파라미터와 검색 공간 설정
    2. 목적 함수에서 XGBoost를 학습 후 예측 성능 결과를 반환 값으로 설정
    3. fmin() 함수에 목적 함수를 하이퍼 파라미터 검색 공간의 입력값들을 사용해 최적의 예측 성능 결과를 반환하는 최적 입력값들을 결정
    - 주의사항
        - HyperOpt는 입력값과 반환값이 모두 실수형이기 때문에 하이퍼 파라미터 입력 시 형변환을 해야 함
        - 목적 함수는 최솟값을 반환할 수 있도록 최적화해야 하기 때문에 성능 값이 클수록 좋은 성능 지표일 경우 -1을 곱해줘야 함
        - cross_val_score()를 적용할 경우 조기 중단이 지원되지 않음
    - 위스콘신 유방암 데이터 세트 예제
        - 유의사항
            - 검색 공간에서 목적 함수로 입력되는 모든 인자들은 실수형 값 → 정수형 하이퍼 파라미터로 설정할 때에는 정수형으로 형변환 해야 함

        - 목적 함수는 최솟값을 반환할 수 있도록 최적화해야 하기 때문에 성능 값이 클수록 좋은 성능 지표일 경우 -1을 곱해줘야 함 (회귀는 작을수록 좋기 때문에 -1 곱할 필요 없음)
    - objective_func()
        - 반환값 : 교차 검증 기반의 평균 정확도

▼ 4.11 스태킹 앙상블

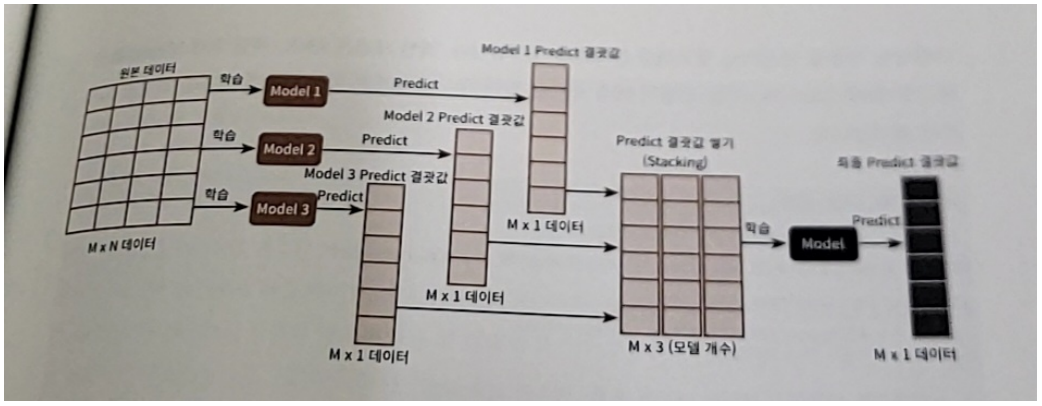
- 스태킹(Stacking) : 개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출
    - 배깅, 부스팅과 공통점을 가짐
    - 가장 큰 차이점 : 개별 알고리즘으로 예측한 데이터를 기반으로 다시 예측 수행
        - 개별 알고리즘의 예측 결과 데이터 세트 : 최종적인 메타 데이터 세트 → 별도의 ML 알고리즘으로 최종 학습 수행 → 테스트 데이터를 기반으로 다시 최종 예측 수행
            - 메타 모델 : 개별 모델의 예측된 데이터 세트를 다시 기반으로 하여 학습하고 예측하는 방식
- 스태킹 모델



    - 개별적인 기반 모델
    - 최종 메타 모델
        - 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어서 학습
    - 핵심 : 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합 → 최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 피처 데이터 세트를 만

드는 것

- 현실 모델에 적용은 많지 않음, 대회에서 조금이라도 성능 수치를 높여야 할 경우 사용
- 많은 개별 모델이 필요
- 일반적으로 최적으로 파라미터를 튜닝한 상태에서 스태킹 모델을 만드는 것이 일반적
- 예시



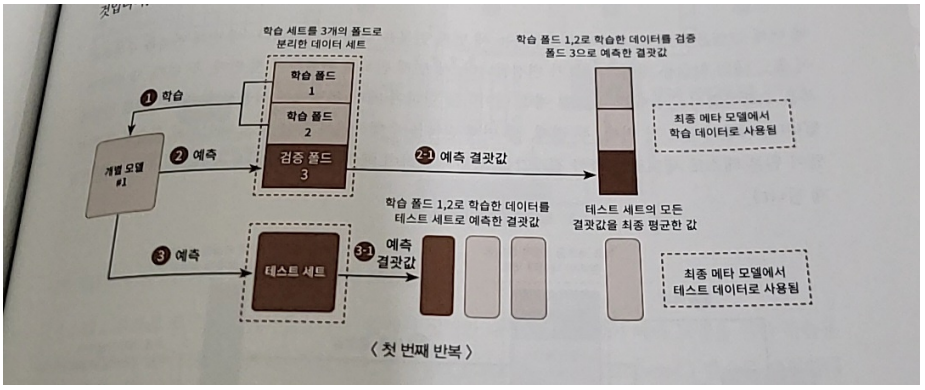
  - M개의 row, N개의 feature를 가진 데이터 세트에 스태킹 앙상블을 적용한다고 가정
  - ML 알고리즘 모델은 모두 3개
    - 각각 모델별로 학습 → 예측 수행 → 각각 M개의 row를 가진 1개의 레이블 값 도출
    - 모델별로 도출된 예측 레이블 값을 다시 합해서(스태킹) 새로운 데이터 세트 만듦 → 최종 모델을 적용해 최종 예측 수행

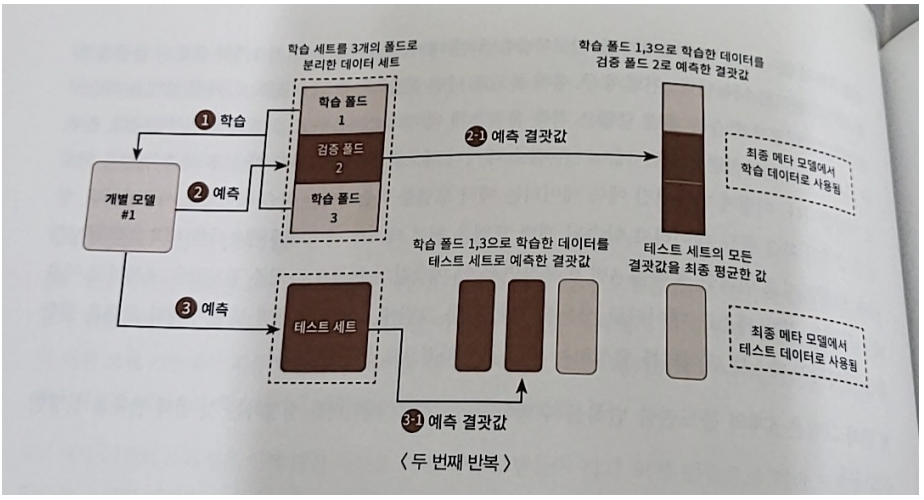
- **기본 스태킹 모델**
  - 위스콘신 암 데이터 세트 예제
    - 학습 데이터 세트와 테스트 데이터 세트로 나눔
    - 스태킹에 사용될 ML 알고리즘 클래스 생성
      - 개별 모델 : KNN, 랜덤 포레스트, 결정 트리, 에이다부스트
      - 학습/예측 최종 모델 : 로지스틱 회귀
    - 개별 모델 학습
    - 개별 모델 예측 데이터 세트 반환, 모델 예측 정확도 확인

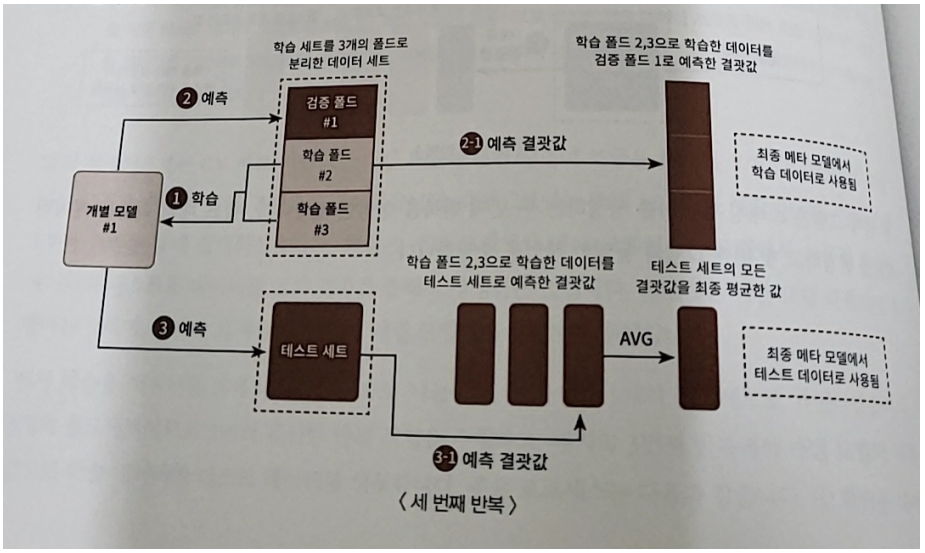
        - 예측값을 피처 값으로 만들어 로지스틱 회귀에서 학습 데이터로 재사용
            - 1차원 ndarray이므로 transpose()를 이용해 행과 열 위치 바꾼 ndarray로 변환하면 됨
        - 로지스틱 회귀로 최종 학습, 예측 정확도 측정
        - ⇒ 정확도가 개별 모델 정확도보다 향상됨 (무조건 개별 모델보다 정확도 좋아지는 건 아님)
- **cv 세트 기반의 스태킹**
    - 과적합 개선 위해 최종 메타 모델을 위한 데이터 세트를 만들 때 교차 검증 기반으로 예측된 결과 데이터 세트를 이용
    - 개별 모델들이 각각 교차 검증으로 메타 모델을 위한 학습용 스태킹 데이터 생성, 예측을 위한 테스트용 스태킹 데이터 생성 → 메타 모델이 학습과 예측 수행
        1. 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로 메타 모델을 위한 학습용/테스트용 데이터 생성
        2. 1단계에서 개별 모델들이 생성한 학습용 데이터를 모두 스태킹으로 합쳐 메타 모델이 학습할 최종 학습용 데이터 세트 생성, 테스트 데이터도 마찬가지로 생성함.

           메타 모델은 최종 생성된 학습 데이터 세트와 원본 학습 데이터의 레이블 데이터를 기반으로 학습한 뒤, 최종적으로 생성된 테스트 데이터 세트를 예측, 원본 테스트 데이터의 레이블 데이터를 기반으로 평가
            - 개별 모델 레벨에서 수행, 로직을 여러 개의 개별 모델에서 동일하게 수행함
            - 먼저 학습용 데이터를 N개의 폴드로 나눔. (예시에서는 3개로 나눔) → 3번의 유사한 반복 작업 수행, 마지막 반복에서 개별 모델의 예측 값으로 학습 데이터와 테스트 데이터 생성
                1. 학습용 데이터로 3개의 폴드로 나눔 → 2개는 학습 위한 데이터 폴드, 1개는 검증 위한 데이터 폴드
                2. 학습 데이터 기반으로 개별 모델 학습 → 검증 폴드 1개 데이터로 예측, 결과 저장 → 이러한 로직 3번 반복, 예측 데이터는 메타 모델 학습 데이터로 사용됨

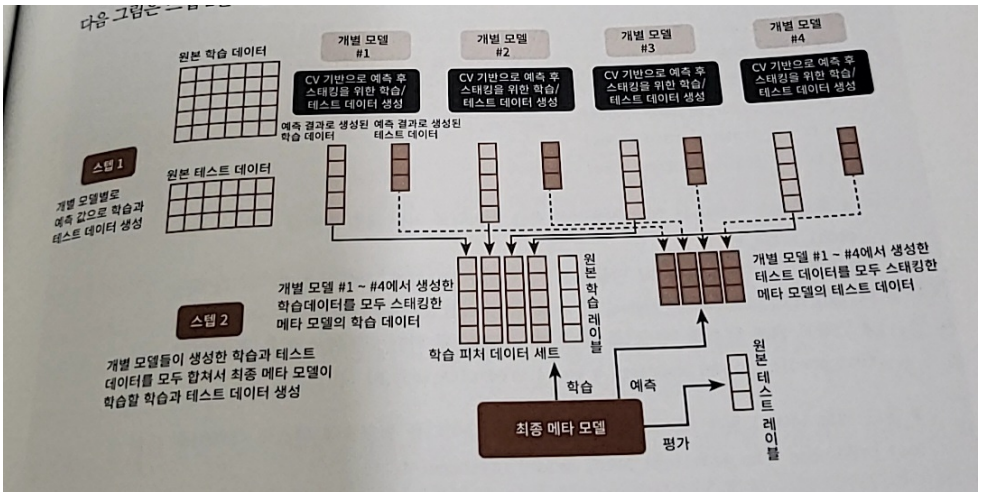
3. 2개의 학습 폴드 데이터로 학습된 개별 모델은 원본 테스트 데이터를 예측하여 예측값 생성 → 로직 3번 반복 → 예측값 평균으로 최종 결괏값 생성, 메타 모델을 위한 테스트 데이터로 사용







- Step 2: 각 모델들이 스텝 1로 생성한 학습, 테스트 데이터를 모두 합치기 → 최종적으로 메타 모델이 사용할 학습 데이터와 테스트 데이터 생성



- 스텝 1, 2 코드 구현
  - get_stacking_base_datasets() 함수 생성 - 메타 모델을 위한 학습용, 테스트 데이터 생성
    - 폴드의 개수만큼 반복 수행, 폴드된 학습용 데이터로 학습 → 예측 결괏값을 기반으로 메타 모델을 위한 학습,테스트용 데이터 새롭게 생성
    - 여러 개의 분류 모델별로 stack_base_model() 함수 수행
    - 모델별로 get_stacking_base_datasets() 함수 호출 → 학습용, 테스트용 데이터 세트 반환
  - 넘파이의 concatenate()를 이용해 각 모델별 학습 데이터와 테스트 데이터 합침
  - Stack_final_X_train : 메타 모델이 학습할 학습용 피처 데이터 세트
  - Stack_final_X_test : 메타 모델이 예측할 테스트용 피처 데이터 세트