2025-2 데이터문제해결및실습1

11주차-1
[경진대회3] 안전 운전자 예측_1

세종대학교
인공지능데이터사이언스학과
박동현 교수

- 본 강의는 골든래빗 출판사의 머신러닝/딥러닝 문제해결전략이 제공하는 강의 교안에 기반함.

Chapter

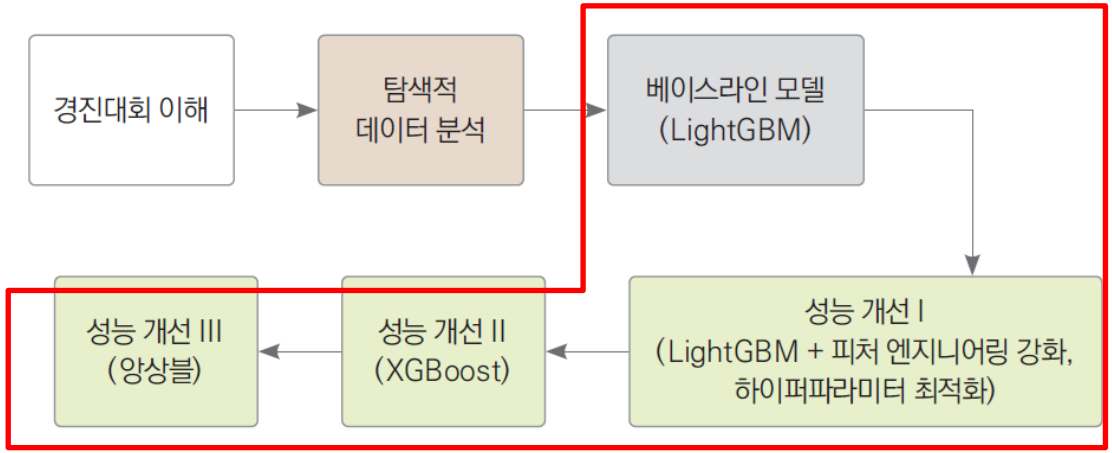
08

[경진대회]
안전 운전자 예측_2

□ 학습 목표

실제 기업 데이터를 활용한 안전 운전자 예측 경진대회에 참가한다. 먼저 탐색적 데이터 분석으로 모델링에 필요 없는 데이터를 찾아본다. 이번에도 베이스라인 모델에서 시작해 성능이 좋은 모델로 발전시킨다. 이 과정에서 캐글에서 실제로 많이 활용되는 여러 가지 고급 모델링 기법을 배울 수
있다.

□ 학습 순서



8.3 베이스라인 모델

8.3.2 평가지표 계산 함수 작성 (설명 업데이트)

지니계수란?

- 경제학에서 지니계수는 소득 불평등 정도를 나타내는 지표.
- 지니계수가 0에 가까울 수록 소득 수준이 평등하고, 1에 가깝게 클수록 불평등하다.

A영역 = 로렌츠 곡선과 완전균형 대각선과의 사이 = 불평등 면적

B영역 = 삼각형 전체면적 - A영역

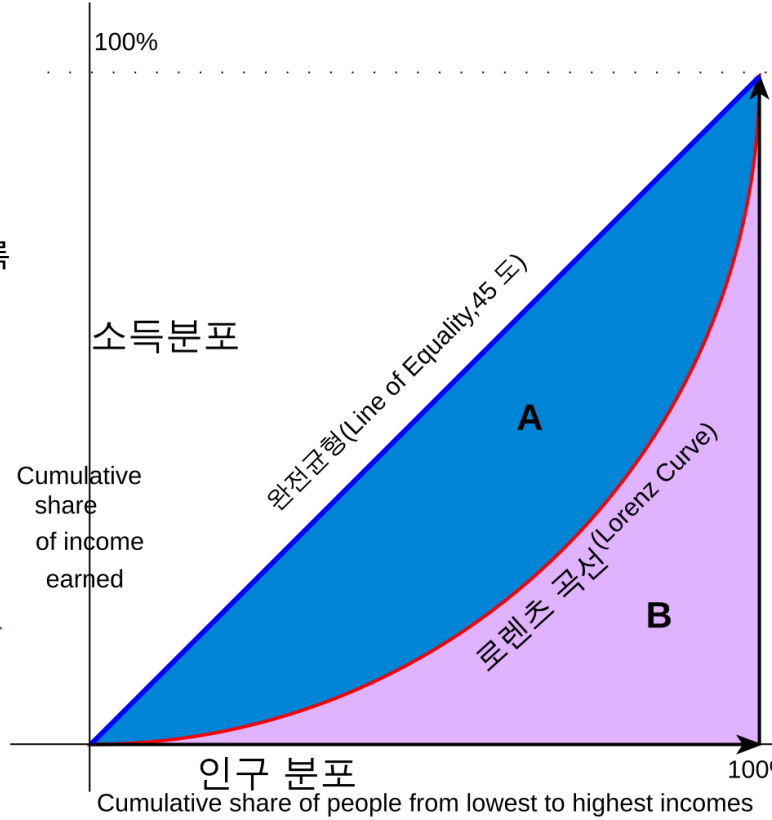
지니계수 $G = \frac{A}{A+B}$

소득 완전평등 = 0 , $G = \frac{0}{0+B}$

로렌츠곡선이 완전균형 대각선에 수렴하여 일치될 때 A영역은 B영역에 의해 없어진다고 볼 수 있다.

소득 완전불평등 = 1 , $G = \frac{A}{A+0}$

로렌츠곡선이 수직선에 수렴하여 일치될 때 B영역은 A영역에 의해 없어진다고 볼 수 있다.



8.3 베이스라인 모델

8.3.2 평가지표 계산 함수 작성 (설명 업데이트)

지니계수란?

- Gini Coefficient (지니계수)는 주로 소득 불평등을 측정하는 지표로 사용되는데, 머신러닝의 변별력 판별에 이용.

- 머신러닝에서 지니계수는 모델의 예측 성능을 측정하는 데 쓰인다.

- 1에 가까울 수록 변별력이 크다고 볼 수 있으며, 0.6 이상이면 모형의 변별력이 뛰어난 것으로 판단할 수 있다. 특히, 타겟 값이 한 쪽으로 치우쳐져 있을 때 이를 구별할 때 많이 쓴다.

- AUC 지표와 관계가 있다. GINI = (2*AUC)−1 와 같다.

8.3 베이스라인 모델

8.3.2 평가지표 계산 함수 작성 (설명 업데이트)

정규화 지니계수 계산 함수

- 정규화란 값의 범위를 0~1 사이로 조정한다는 뜻이므로, **정규화** 지니계수는 값이 0에 가까울수록 성능이 나쁘고, 1에 가까울수록 성능이 좋다.

$$\text{정규화 지니계수} = \frac{\text{예측 값에 대한 지니계수}}{\text{예측이 완벽할 때의 지니계수}}$$

- 예측값에 대한 지니계수: 예측값과 실젯값으로 구한 지니계수
- 예측이 완벽할 때의 지니계수: 실젯값과 실젯값으로 구한 지니계수

8.3 베이스라인 모델

8.3.2 평가지표 계산 함수 작성 (설명 업데이트)

정규화 지니계수 계산 함수

```
import numpy as np

def eval_gini(y_true, y_pred):
    # 실제값과 예측값의 크기가 같은지 확인 (값이 다르면 오류 발생)
    assert y_true.shape == y_pred.shape

    n_samples = y_true.shape[0]                        # 데이터 개수
    L_mid = np.linspace(1 / n_samples, 1, n_samples) # 대각선 값

    # 1) 예측값에 대한 지니계수
    pred_order = y_true[y_pred.argsort()] # y_pred 크기순으로 y_true 값 정렬 (예측값과 실제값)
    L_pred = np.cumsum(pred_order) / np.sum(pred_order) # 로렌츠 곡선
    G_pred = np.sum(L_mid - L_pred)        # 예측 값에 대한 지니계수
    # 2) 예측이 완벽할 때 지니계수
    true_order = y_true[y_true.argsort()] # y_true 크기순으로 y_true 값 정렬 (실제값과 실제값)
    L_true = np.cumsum(true_order) / np.sum(true_order) # 로렌츠 곡선
    G_true = np.sum(L_mid - L_true)        # 예측이 완벽할 때 지니계수

    # 정규화된 지니계수
    return G_pred / G_true

def gini(preds, dtrain):
    labels = dtrain.get_label()
    return 'gini', eval_gini(labels, preds), True # 반환값
```

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증 (설명 업데이트)

OOF (Out-Of-Fold) 예측 방식

- K 폴드 교차 검증을 수행하면서 각 폴드마다 테스트 데이터로 예측하는 방식
- OFF 예측 절차

1. 전체 훈련 데이터를 K개 그룹나눈다.
2. K개 그룹 중 한 그룹은 검증 데이터, 나머지 K-1개 그룹은 훈련 데이터로 지정한다.
3. 훈련 데이터로 모델을 훈련한다.
4. 훈련된 모델을 이용해 검증 데이터로 타깃 확률을 예측하고, **전체 테스트 데이터로도 타깃 확률을 예측한다.**
5. 검증 데이터로 구한 예측 확률과 테스트 데이터로 구한 예측 확률을 기록한다.
6. 검증 데이터를 다른 그룹으로 바꿔가며 2~5번 절차를 총 K번 반복한다.
7. K개 그룹의 검증 데이터로 예측한 확률을 훈련 데이터 실제 타깃값과 비교해 성능 평가점수를 계산한다. 이 점수로 모델 성능을 가늠해볼 수 있습니다.
8. **테스트 데이터로 구한 K개 예측 확률의 평균을 구한다. 이 값이 최종 예측 확률이며, 제출해야 하는 값이다**

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증 (설명 업데이트)

OOF 예측 방식

- OOF 예측 방식의 장점
  - 과대적합 방지에 효과적이다.
  - 앙상블 효과가 있어 모델 성능이 좋아진다.

$$\frac{\left[\text{테스트 데이터 예측 확률 1} + \text{테스트 데이터 예측 확률 2} + \text{테스트 데이터 예측 확률 3} + \text{테스트 데이터 예측 확률 4} + \text{테스트 데이터 예측 확률 5}\right]}{5} = \text{최종 예측 확률}$$

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증 (설명 업데이트)

OOF 방식으로 LightGBM 훈련

- 층화 K 폴드 교차 검증기 생성하기

```
from sklearn.model_selection import StratifiedKFold

# 층화 K 폴드 교차 검증기
folds = StratifiedKFold(n_splits=5, shuffle=True, random_state=1991)
```

- LightGBM의 하이퍼파라미터를 설정하기

```
params = {'objective': 'binary',
'learning_rate': 0.01,
'force_row_wise': True,
'random_state': 0}
```

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증 (설명 업데이트)

OOF 방식으로 LightGMB 훈련

- 1차원 배열 두 개 만들기

```
f# OOF 방식으로 훈련된 모델로 검증 데이터 타깃값을 예측한 확률을 담을 1차원 배열
oof_val_preds = np.zeros(X.shape[0])
# OOF 방식으로 훈련된 모델로 테스트 데이터 타깃값을 예측한 확률을 담을 1차원 배열
oof_test_preds = np.zeros(X_test.shape[0])
```

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증 (설명 업데이트)

OOF 방식으로 LightGMB 훈련

- LightGBM 모델을 훈련하기

```
import lightgbm as lgb

# OOF 방식으로 모델 훈련, 검증, 예측
for idx, (train_idx, valid_idx) in enumerate(folds.split(X, y)): #1
    # 각 폴드를 구분하는 문구 출력
    print('#'*40, f'폴드 {idx+1} / 폴드 {folds.n_splits}', '#'*40)

    # 훈련용 데이터, 검증용 데이터 설정 #2
    X_train, y_train = X[train_idx], y[train_idx] # 훈련용 데이터
    X_valid, y_valid = X[valid_idx], y[valid_idx] # 검증용 데이터

    # LightGBM 전용 데이터셋 생성 #3
    dtrain = lgb.Dataset(X_train, y_train) # LightGBM 전용 훈련 데이터셋
    dvalid = lgb.Dataset(X_valid, y_valid) # LightGBM 전용 검증 데이터셋
```

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증

OOF 방식으로 LightGMB 훈련

- LightGBM 모델을 훈련하기

```
# LightGBM 모델 훈련 #4
lgb_model = lgb.train(params=params, # 훈련용 하이퍼파라미터
                      train_set=dtrain, # 훈련 데이터셋
                      num_boost_round=1000, # 부스팅 반복 횟수
                      valid_sets=dvalid, # 성능 평가용 검증 데이터셋
                      feval=gini, # 검증용 평가지표
                      early_stopping_rounds=100, # 조기종료 조건
                      verbose_eval=100) # 100번째마다 점수 출력


# 테스트 데이터를 활용해 OOF 예측 #5
oof_test_preds += lgb_model.predict(X_test)/folds.n_splits

# 모델 성능 평가를 위한 검증 데이터 타깃값 예측 #6
oof_val_preds[valid_idx] += lgb_model.predict(X_valid)

# 검증 데이터 예측 확률에 대한 정규화 지니계수 #7
gini_score = eval_gini(y_valid, oof_val_preds[valid_idx])
print(f'폴드 {idx+1} 지니계수: {gini_score}\n')
```

8.3 베이스라인 모델

8.3.3 모델 훈련 및 성능 검증

OOF 방식으로 LightGMB 훈련

- 모델을 훈련할 때 출력된 로그를 살펴보기

```
######################################## 폴드 5 / 폴드 5 ########################
################
[LightGBM] [Info] Number of positive: 17355, number of negative: 458815
[LightGBM] [Info] Total Bins 1098
[LightGBM] [Info] Number of data points in the train set: 476170, number of used
features: 200
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.036447 -> initscore=-3.274766
[LightGBM] [Info] Start traini                                    4766
Training until validation scores don't improve for 100 rounds
[100] valid_0's binary_logloss: 0.153483      valid_0's gini: 0.262106
[200] valid_0's binary_logloss: 0.152646      valid_0's gini: 0.273406
[300] valid_0's binary_logloss: 0.152291      valid_0's gini: 0.279805
[400] valid_0's binary_logloss: 0.152093      valid_0's gini: 0.284645
[5    0's binary_logloss: 0.152004      valid_0's gini: 0.28713
[600] valid_0's binary_logloss: 0.151982      valid_0's gini: 0.287668
Early stopping, best iteration is:
[583] valid_0's binary_logloss: 0.15198       valid_0's gini: 0.287804
폴드 5 지니계수 : 0.2878042213842625
```

❶ 성능 평가점수가 두 개

❷ 조기 종료

❸ 종료 시까지 최고 점수를 기록한 이터레이션과 점수

8.3 베이스라인 모델

8.3.4 예측 및 결과 제출

- 최종 예측 확률은 oof_test_preds에 담겨 있다. OOF 예측 방식으로 총 5개 폴드로 교차 검증한 확률값들의 평균이다.

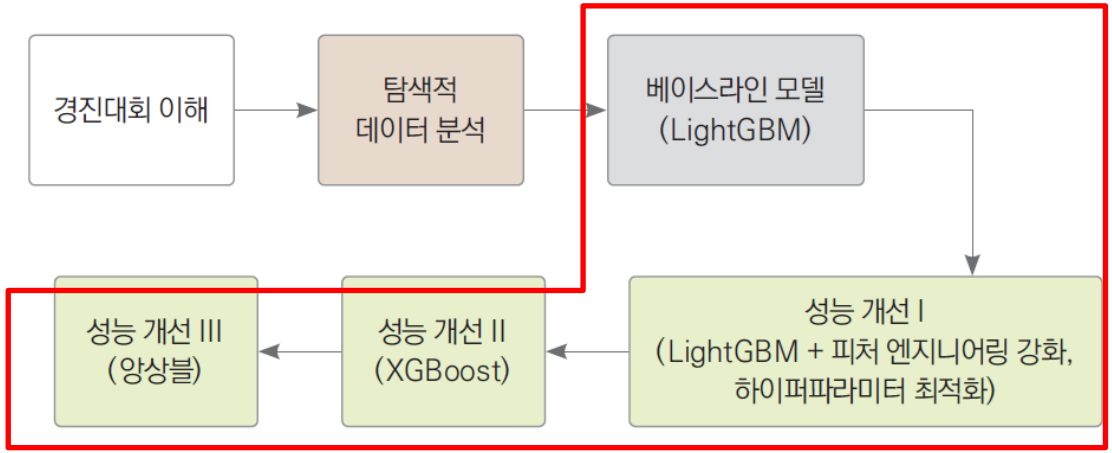
```
submission['target'] = oof_test_preds
submission.to_csv('submission.csv')
```

- 커밋 후 제출한다.

□ 학습 목표

실제 기업 데이터를 활용한 안전 운전자 예측 경진대회에 참가한다. 먼저 탐색적 데이터 분석으로 모델링에 필요 없는 데이터를 찾아본다. 이번에도 베이스라인 모델에서 시작해 성능이 좋은 모델로 발전시킨다. 이 과정에서 캐글에서 실제로 많이 활용되는 여러 가지 고급 모델링 기법을 배울 수
있다.

□ 학습 순서

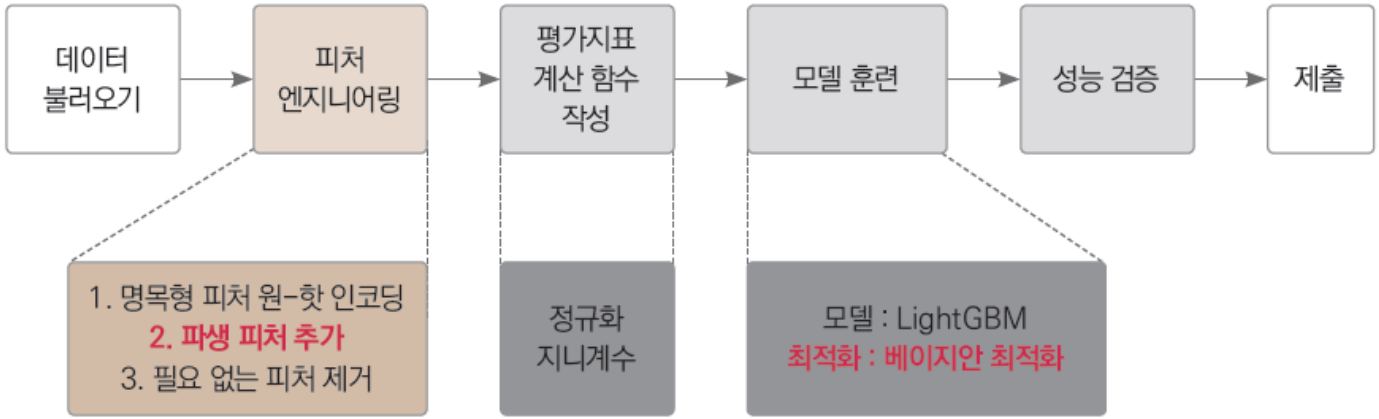


□ 핵심 요약

- LightGBM : 마이크로소프트가 개발한 모델로, 빠르면서 성능이 좋아서 캐글에서 많이 사용한다.
- XGBoost : 성능이 우수한 트리 기반 부스팅 알고리즘. 결정 트리를 병렬로 배치하는 랜덤 포레스트와 달리 직렬로 배치해 사용한다.
- **앙상블** : 여러 모델에서 얻은 예측 결과를 결합해 더 좋은 예측값을 도출하는 방식. 단순하면서 효과가 좋다.

8.4 성능 개선 I : LightGBM 모델

첫 번째 성능 개선으로 베이스라인 모델과 같은 LightGBM을 그대로 사용하되 피처 엔지니어링과 하이퍼파라미터 최적화를 추가로 적용해본다.



8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

데이터 합치기

```
all_data = pd.concat([train, test], ignore_index=True)
all_data = all_data.drop('target', axis=1) # 타깃값 제거

all_features = all_data.columns # 전체 피처
```

명목형 피처 원-핫 인코딩

```
from sklearn.preprocessing import OneHotEncoder

# 명목형 피처
cat_features = [feature for feature in all_features if 'cat' in feature]

# 원-핫 인코딩 적용
onehot_encoder = OneHotEncoder()
encoded_cat_matrix = onehot_encoder.fit_transform(all_data[cat_features])
```

8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

파생 피처 추가

첫 번째! 한 데이터가 가진 결측값 개수를 파생 피처로 만들기

```
# '데이터 하나당 결측값 개수'를 파생 피처로 추가
all_data['num_missing'] = (all_data==-1).sum(axis=1)
```

방금 생성한 파생 피처 num_missing도 remaining_features에 추가

```
# 명목형 피처, calc 분류의 피처를 제외한 피처
remaining_features = [feature for feature in all_features
                      if ('cat' not in feature and 'calc' not in feature)]
# num_missing을 remaining_features에 추가
remaining_features.append('num_missing')
```

8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

파생 피처 추가

두 번째! mix_ind 피처를 만든다. 먼저 분류가 ind인 피처들을 추출한다. 그런 다음 이 피처들을 순회하면서 모든 값을 연결한다.

```
# 분류가 ind인 피처
ind_features = [feature for feature in all_features if 'ind' in feature]

is_first_feature = True
for ind_feature in ind_features:
    if is_first_feature:
       all_data['mix_ind'] = all_data[ind_feature].astype(str) + '_'
       is_first_feature = False
    else:
       all_data['mix_ind'] += all_data[ind_feature].astype(str) + '_'
```

새로운 피처 mix_ind 만들기

```
all_data['mix_ind']
```

8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

파생 피처 추가

세 번째! 명목형 피처의 고윳값별 개수를 새로운 피처로 추가한다. 고윳값별 개수는 value_counts()로 구한다. 예를 들어 ps_ind_02_cat 피처의 고윳값별 개수는 다음 코드로 확인한다. 순회하면서 모든 값을 연결한다.

```
all_data['ps_ind_02_cat'].value_counts()
```

Series 타입을 딕셔너리 타입으로 바꾸려면 to_dict( )를 호출한다.

```
all_data['ps_ind_02_cat'].value_counts().to_dict()
```

8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

파생 피처 추가

명목형 피처의 고윳값별 개수를 파생 피처로 만들어본다. 이전에 추가한 mix_ind 피처도 명목형 피처이므로, cat 분류에 속하는 피처들(cat_features)과 mix_ind 피처를 모두 명목형 피처로 간주한다.

```
cat_count_features = []
for feature in cat_features+['mix_ind']:
val_counts_dict = all_data[feature].value_counts().to_dict()
all_data[f'{feature}_count'] = all_data[feature].apply(lambda x:
val_counts_dict[x])
cat_count_features.append(f'{feature}_count')
```

새로 추가한 피처명을 보자. cat_count_features에 방금 추가한 새로운 피처명이 들어 있습니다.

```
all_data['ps_ind_02_cat'].value_counts().to_dict()
```

8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

필요 없는 피처 제거

탐색적 데이터 분석에서 필요 없다고 판단한 피처는 제거한 다음(분석 정리 5~8), 지금까지 피처 엔지니어링한 모든 데이터를 합친다.

```
from scipy import sparse
# 필요 없는 피처들
drop_features = ['ps_ind_14', 'ps_ind_10_bin', 'ps_ind_11_bin',
                 'ps_ind_12_bin', 'ps_ind_13_bin', 'ps_car_14']

# remaining_features, cat_count_features에서 drop_features를 제거한 데이터
all_data_remaining = all_data[remaining_features+cat_count_features].drop(drop_
features, axis=1)

# 데이터 합치기
all_data_sprs = sparse.hstack([sparse.csr_matrix(all_data_remaining),
                               encoded_cat_matrix],
                               format='csr')
```

8.4 성능 개선 I : LightGBM 모델

8.4.1 피처 엔지니어링

데이터 나누기

피처 엔지니어링을 마쳤으니 다시 훈련 데이터와 테스트 데이터로 나눈다.

```
num_train = len(train) # 훈련 데이터 개수

# 훈련 데이터와 테스트 데이터 나누기
X = all_data_sprs[:num_train]
X_test = all_data_sprs[num_train:]

y = train['target'].values
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 하이퍼파라미터 최적화

데이터셋 준비

베이지안 최적화를 위한 데이터셋을 만든다.

```
import lightgbm as lgb
from sklearn.model_selection import train_test_split

# 8:2 비율로 훈련 데이터, 검증 데이터 분리 (베이지안 최적화 수행용)
X_train, X_valid, y_train, y_valid = train_test_split(X, y, # 1
                                                      test_size=0.2,
                                                      random_state=0)

# 베이지안 최적화용 데이터셋 2
bayes_dtrain = lgb.Dataset(X_train, y_train)
bayes_dvalid = lgb.Dataset(X_valid, y_valid)
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 하이퍼파라미터 최적화

하이퍼파라미터 범위 설정

하이퍼파라미터 범위를 설정하는 방법은 두 가지가 있다.

1. 하이퍼파라미터 범위를 점점 좁히는 방법
2. 다른 상위권 캐글러가 설정한 하이퍼파라미터를 참고하는 방법

```
# 베이지안 최적화를 위한 하이퍼파라미터 범위
param_bounds = {'num_leaves': (30, 40),
'lambda_l1': (0.7, 0.9),
'lambda_l2': (0.9, 1),
'feature_fraction': (0.6, 0.7),
'bagging_fraction': (0.6, 0.9),
'min_child_samples': (6, 10),
'min_child_weight': (10, 40)}

# 값이 고정된 하이퍼파라미터
fixed_params = {'objective': 'binary', # 1
'learning_rate': 0.005, # 2
'bagging_freq': 1, # 3
'force_row_wise': True, # 4
'random_state': 1991} # 5
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 하이퍼파라미터 최적화

(베이지안 최적화용) 평가지표 계산 함수 작성

```
def eval_function(num_leaves, lambda_l1, lambda_l2, feature_fraction,
                  bagging_fraction, min_child_samples, min_child_weight):
    '''최적화하려는 평가지표(지니계수) 계산 함수'''

    # 베이지안 최적화를 수행할 하이퍼파라미터 1
    params = {'num_leaves': int(round(num_leaves)),
              'lambda_l1': lambda_l1,
              'lambda_l2': lambda_l2,
              'feature_fraction': feature_fraction,
              'bagging_fraction': bagging_fraction,
              'min_child_samples': int(round(min_child_samples)),
              'min_child_weight': min_child_weight,
              'feature_pre_filter': False}
    # 고정된 하이퍼파라미터도 추가 2
    params.update(fixed_params)

    print('하이퍼파라미터:', params)
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 하이퍼파라미터 최적화

(베이지안 최적화용) 평가지표 계산 함수 작성

```
    # LightGBM 모델 훈련 3
    lgb_model = lgb.train(params=params,
                          train_set=bayes_dtrain,
                          num_boost_round=2500,
                          valid_sets=bayes_dvalid,
                          feval=gini,
                          early_stopping_rounds=300,
                          verbose_eval=False)
    # 검증 데이터로 예측 수행 4
    preds = lgb_model.predict(X_valid)
    # 지니계수 계산 5
    gini_score = eval_gini(y_valid, preds)
    print(f'지니계수 : {gini_score}\n')

    return gini_score # 6
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 하이퍼파라미터 최적화

최적화 수행

하이퍼파라미터 범위와 평가지표 계산 함수를 만들었으니, 베이지안 최적화 객체를 생성한다.

생성 파라미터로 eval_function과 param_bounds를 전달한다.

```
from bayes_opt import BayesianOptimization

# 베이지안 최적화 객체 생성
optimizer = BayesianOptimization(f=eval_function, # 평가지표 계산 함수
                                 pbounds=param_bounds, # 하이퍼파라미터 범위
                                 random_state=0)
```

다음으로 maximize() 메서드를 호출해 베이지안 최적화를 수행한다.

```
# 베이지안 최적화 수행
optimizer.maximize(init_points=3, n_iter=6)
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 하이퍼파라미터 최적화

결과 확인

최적화가 끝나면 지니계수가 최대가 되는 하이퍼파라미터, 즉 최적 하이퍼파라미터를 출력해본다.

```
# 평가함수 점수가 최대일 때 하이퍼파라미터
max_params = optimizer.max['params']
max_params
```

이중 num_leaves와 min_child_samples는 원래 정수형 하이퍼파라미터이므로 정수형으로 변환하여 다시 저장한다.

```
# 정수형 하이퍼파라미터 변환
max_params['num_leaves'] = int(round(max_params['num_leaves']))
max_params['min_child_samples'] = int(round(max_params['min_child_samples']))
```

고정된 파라미터들(fixed_params)도 추가한다.

```
# 값이 고정된 하이퍼파라미터 추가
max_params.update(fixed_params)
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 모델 훈련 및 성능 검증

```
from sklearn.model_selection import StratifiedKFold

# 층화 K 폴드 교차 검증기 생성
folds = StratifiedKFold(n_splits=5, shuffle=True, random_state=1991)

# OOF 방식으로 훈련된 모델로 검증 데이터 타깃값을 예측한 확률을 담을 1차원 배열
oof_val_preds = np.zeros(X.shape[0])
# OOF 방식으로 훈련된 모델로 테스트 데이터 타깃값을 예측한 확률을 담을 1차원 배열
oof_test_preds = np.zeros(X_test.shape[0])

# OOF 방식으로 모델 훈련, 검증, 예측
for idx, (train_idx, valid_idx) in enumerate(folds.split(X, y)):
    # 각 폴드를 구분하는 문구 출력
    print('#'*40, f'폴드 {idx+1} / 폴드 {folds.n_splits}', '#'*40)

    # 훈련용 데이터, 검증용 데이터 설정
    X_train, y_train = X[train_idx], y[train_idx] # 훈련용 데이터
    X_valid, y_valid = X[valid_idx], y[valid_idx] # 검증용 데이터
```

8.4 성능 개선 I : LightGBM 모델

8.4.2 모델 훈련 및 성능 검증

```
# LightGBM 전용 데이터셋 생성
dtrain = lgb.Dataset(X_train, y_train) # LightGBM 전용 훈련 데이터셋
dvalid = lgb.Dataset(X_valid, y_valid) # LightGBM 전용 검증 데이터셋

# LightGBM 모델 훈련
lgb_model = lgb.train(params=max_params, # 최적 하이퍼파라미터 1
                      train_set=dtrain, # 훈련 데이터셋
                      num_boost_round=2500, # 부스팅 반복 횟수
                      valid_sets=dvalid, # 성능 평가용 검증 데이터셋
                      feval=gini, # 검증용 평가지표
                      early_stopping_rounds=300, # 조기종료 조건
                      verbose_eval=100) # 100번째마다 점수 출력

# 테스트 데이터를 활용해 OOF 예측
oof_test_preds += lgb_model.predict(X_test)/folds.n_splits
# 모델 성능 평가를 위한 검증 데이터 타깃값 예측
oof_val_preds[valid_idx] += lgb_model.predict(X_valid)

# 검증 데이터 예측 확률에 대한 정규화 지니계수
gini_score = eval_gini(y_valid, oof_val_preds[valid_idx])
print(f'폴드 {idx+1} 지니계수 : {gini_score}\n')
```

8.4 성능 개선 I : LightGBM 모델

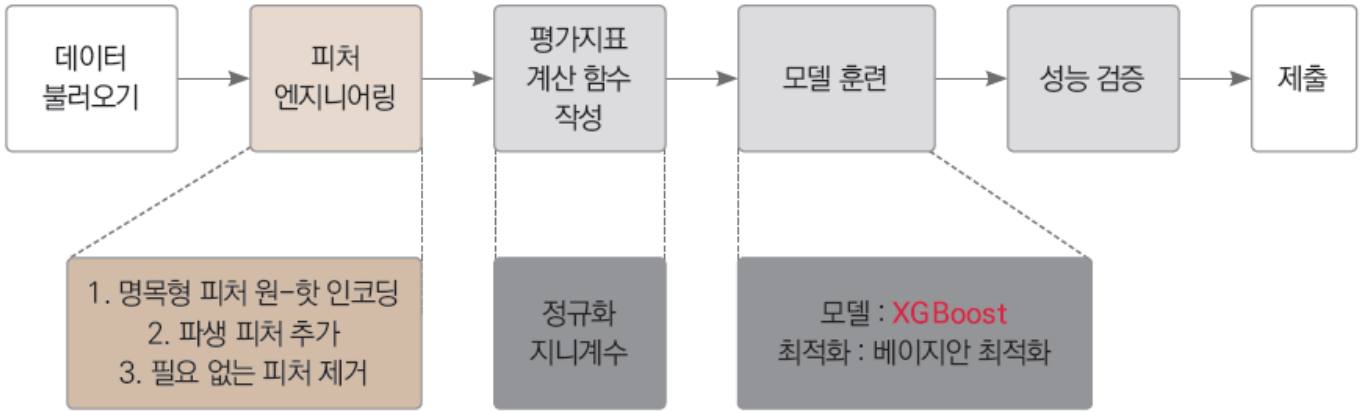
8.4.4 예측 및 결과 제출

최종 예측 확률인 oof_test_preds를 활용해 제출 파일을 만든다. 다음 코드를 실행한뒤 커밋 후 제출한다.

```
submission['target'] = oof_test_preds
submission.to_csv('submission.csv')
```

8.5 성능 개선 II : XGBoost 모델

모델을 XGBoost로 바꿔본다. XGBoost는 성능이 우수한 트리 기반 부스팅 알고리즘으로, 결정 트리를 병렬로 배치하는 랜덤 포레스트와 달리 직렬로 배치해 사용한다.



앞서 사용한 코드를 XGBoost 기반으로 바꾸려면 다음을 수정해야 한다.

1. 지니계수 반환값
2. 데이터셋 객체
3. 모델 하이퍼파라미터명

8.5 성능 개선 II : XGBoost 모델

8.5.1 피처 엔지니어링

XGBoost용 지니계수 계산 함수는 반환값이 두 개다. 평가지표명과 평가점수만 반환한다. 사라진 '평가점수가 높으면 좋은지 여부'는 XGBoost 모델 객체의 train( ) 메서드에 따로 전달해야 한다.

```
# XGBoost용 gini() 함수
def gini(preds, dtrain):
    labels = dtrain.get_label()
    return 'gini', eval_gini(labels, preds)
```

8.5 성능 개선 II : XGBoost 모델

8.5.2 하이퍼파라미터 최적화

데이터셋 준비

베이지안 최적화 수행용 데이터셋을 만든다. LightGBM은 lgb.Dataset( )으로 데이터셋을 만든다. 반면 XGBoost는 xgb.DMatrix( )로 만든다.

```
import xgboost as xgb
from sklearn.model_selection import train_test_split

# 8:2 비율로 훈련 데이터, 검증 데이터 분리 (베이지안 최적화 수행용)
X_train, X_valid, y_train, y_valid = train_test_split(X, y,
                                                      test_size=0.2,
                                                      random_state=0)

# 베이지안 최적화용 데이터셋
bayes_dtrain = xgb.DMatrix(X_train, y_train)
bayes_dvalid = xgb.DMatrix(X_valid, y_valid)
```

8.5 성능 개선 II : XGBoost 모델

8.5.2 하이퍼파라미터 최적화

하이퍼파라미터 범위 설정

```
# 베이지안 최적화를 위한 하이퍼파라미터 범위
param_bounds = {'max_depth': (4, 8),
                'subsample': (0.6, 0.9),
                'colsample_bytree': (0.7, 1.0),
                'min_child_weight': (5, 7),
                'gamma': (8, 11),
                'reg_alpha': (7, 9),
                'reg_lambda': (1.1, 1.5),
                'scale_pos_weight': (1.4, 1.6)}

# 값이 고정된 하이퍼파라미터
fixed_params = {'objective': 'binary:logistic',
                'learning_rate': 0.02,
                'random_state': 1991}
```

8.5 성능 개선 II : XGBoost 모델

8.5.2 하이퍼파라미터 최적화

(베이지안 최적화용) 평가지표 계산 함수 작성

이제 베이지안 최적화에 사용하기 위한 eval_function( ) 함수를 살펴본다. 이 함수는 XGBoost 하이퍼파라미터를 인수로 받아서 XGBoost를 훈련한 뒤 평가지표인 지니계수를 반환한다. 큰 흐름은 LightGBM용 eval_function( )과 유사하나, 다른 점은 다음과 같다.

1. 하이퍼파라미터명
2. train( ) 메서드 내 검증 데이터 전달 방식
3. train( ) 메서드 내 maximize 파라미터
4. predict( ) 메서드에 DMatrix 타입을 전달하는 점

8.5 성능 개선 II : XGBoost 모델

8.5.2 하이퍼파라미터 최적화

(베이지안 최적화용) 평가지표 계산 함수 작성

```
def eval_function(max_depth, subsample, colsample_bytree, min_child_weight,
                  reg_alpha, gamma, reg_lambda, scale_pos_weight):
    '''최적화하려는 평가지표(지니계수) 계산 함수'''
    # 베이지안 최적화를 수행할 하이퍼파라미터 1
    params = {'max_depth': int(round(max_depth)),
              'subsample': subsample,
              'colsample_bytree': colsample_bytree,
              'min_child_weight': min_child_weight,
              'gamma': gamma,
              'reg_alpha':reg_alpha,
              'reg_lambda': reg_lambda,
              'scale_pos_weight': scale_pos_weight}
# 값이 고정된 하이퍼파라미터도 추가
params.update(fixed_params)

print('하이퍼파라미터 :', params)

# XGBoost 모델 훈련 2
xgb_model = xgb.train(params=params,
                      dtrain=bayes_dtrain,
```

8.5 성능 개선 II : XGBoost 모델

8.5.2 하이퍼파라미터 최적화

(베이지안 최적화용) 평가지표 계산 함수 작성

```
                      num_boost_round=2000,
                      evals=[(bayes_dvalid, 'bayes_dvalid')], # 3
                      maximize=True, # 4
                      feval=gini,
                      early_stopping_rounds=200,
                      verbose_eval=False)
best_iter = xgb_model.best_iteration # 최적 반복 횟수 5
# 검증 데이터로 예측 수행 6
preds = xgb_model.predict(bayes_dvalid, # 7
                      iteration_range=(0, best_iter)) # 8

# 지니계수 계산
gini_score = eval_gini(y_valid, preds)
print(f'지니계수 : {gini_score}\n')

return gini_score
```

8.5 성능 개선 II : XGBoost 모델

8.5.2 하이퍼파라미터 최적화

최적화 수행

```
from bayes_opt import BayesianOptimization

# 베이지안 최적화 객체 생성
optimizer = BayesianOptimization(f=eval_function,
                                 pbounds=param_bounds,
                                 random_state=0)
# 베이지안 최적화 수행
optimizer.maximize(init_points=3, n_iter=6)
```

결과 확인

```
# 평가함수 점수가 최대일 때 하이퍼파라미터
max_params = optimizer.max['params']
max_params

# 정수형 하이퍼파라미터 변환
max_params['max_depth'] = int(round(max_params['max_depth']))
# 값이 고정된 하이퍼파라미터 추가
max_params.update(fixed_params)
max_params
```

8.5 성능 개선 II : XGBoost 모델

8.5.3 모델 훈련 및 성능 검증

최적 하이퍼파라미터를 이용해 XGBoost 모델을 훈련해본다. 역시 OOF 방식으로 예측한다.

```
from sklearn.model_selection import StratifiedKFold

# 층화 K 폴드 교차 검증기 생성
folds = StratifiedKFold(n_splits=5, shuffle=True, random_state=1991)

# OOF 방식으로 훈련된 모델로 검증 데이터 타깃값을 예측한 확률을 담을 1차원 배열
oof_val_preds = np.zeros(X.shape[0])

# OOF 방식으로 훈련된 모델로 테스트 데이터 타깃값을 예측한 확률을 담을 1차원 배열
oof_test_preds = np.zeros(X_test.shape[0])

# OOF 방식으로 모델 훈련, 검증, 예측
for idx, (train_idx, valid_idx) in enumerate(folds.split(X, y)):
    # 각 폴드를 구분하는 문구 출력
    print('#'*40, f'폴드 {idx+1} / 폴드 {folds.n_splits}', '#'*40)

    # 훈련용 데이터, 검증용 데이터 설정
    X_train, y_train = X[train_idx], y[train_idx]
    X_valid, y_valid = X[valid_idx], y[valid_idx]
```

8.5 성능 개선 II : XGBoost 모델

8.5.3 모델 훈련 및 성능 검증

```
# XGBoost 전용 데이터셋 생성
dtrain = xgb.DMatrix(X_train, y_train)
dvalid = xgb.DMatrix(X_valid, y_valid)
dtest = xgb.DMatrix(X_test)
# XGBoost 모델 훈련
xgb_model = xgb.train(params=max_params,
                      dtrain=dtrain,
                      num_boost_round=2000,
                      evals=[(dvalid, 'valid')],
                      maximize=True,
                      feval=gini,
                      early_stopping_rounds=200,
                      verbose_eval=100)
# 모델 성능이 가장 좋을 때의 부스팅 반복 횟수 저장
best_iter = xgb_model.best_iteration
# 테스트 데이터를 활용해 OOF 예측
oof_test_preds += xgb_model.predict(dtest,
                          iteration_range=(0, best_iter))/folds.n_splits
# 모델 성능 평가를 위한 검증 데이터 타깃값 예측
oof_val_preds[valid_idx] += xgb_model.predict(dvalid,
                                    iteration_range=(0, best_iter))
# 검증 데이터 예측 확률에 대한 정규화 지니계수
gini_score = eval_gini(y_valid, oof_val_preds[valid_idx])
print(f'폴드 {idx+1} 지니계수 : {gini_score}\n')
```

8.5 성능 개선 II : XGBoost 모델

8.5.4 예측 및 결과 제출

최종 예측 확률을 제출해서 프라이빗 점수를 살펴본다.

```
submission['target'] = oof_test_preds
submission.to_csv('submission.csv')
```

8.6 성능 개선 III : LightGBM과 XGBoost 앙상블

여러 모델에서 얻은 예측 결과를 결합해 더 좋은 예측값을 도출하는 방식을 앙상블(ensemble)이라고 한다. 앙상블은 캐글에서 자주 쓰는 기법이다. 상위권을 기록한 많은 캐글러가 앙상블을 사용한다. 방법은 간단하지만 효과는 강력하다. 앙상블 기법을 활용해 점수를 더 높여보자.

8.6.1 앙상블 수행

최종 예측 확률을 제출해서 프라이빗 점수를 살펴본다.

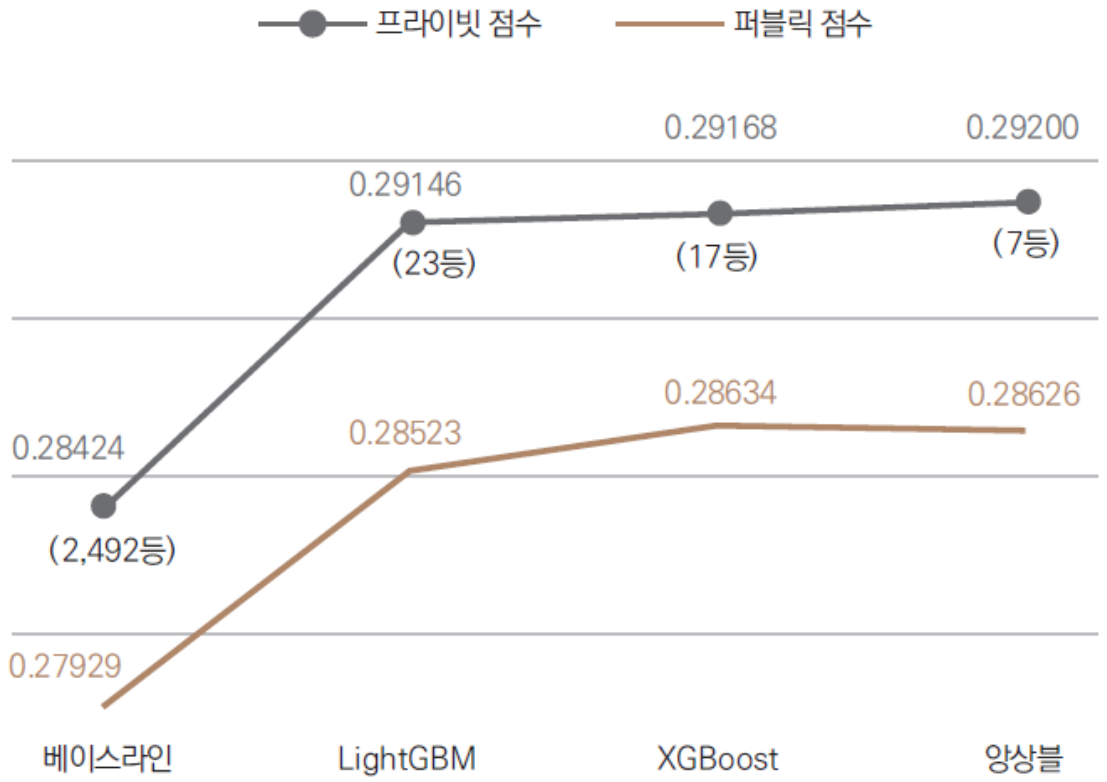
```
oof_test_preds = oof_test_preds_lgb * 0.5 + oof_test_preds_xgb * 0.5
```

8.6.2 예측 및 결과 제출

제출용 파일을 만든다.

```
submission['target'] = oof_test_preds
submission.to_csv('submission.csv')
```

모델별 점수


프라이빗 점수
퍼블릭 점수
0.29168
0.29200
0.29146
(23등)
(17등)
(7등)
0.28634
0.28626
0.28523
0.28424
(2,492등)
0.27929
베이스라인
LightGBM
XGBoost
앙상블