

분류 실습

×

스태킹 앙상블

Bitamin 4 조

김회인 | 김지나 | 문윤지 | 유승길

# × Index

01

분류실습 예제

캐글 신용카드 사기 검출

02

분류모델 실습

Mobile Phone Price Classification

03

스태킹 앙상블

04

스태킹 앙상블 실습

Cross Validation (K-Fold) 기반  
스태킹 모델 실습

# 01 분류실습 예제

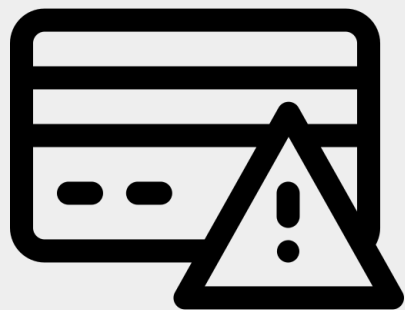
캐글 신용카드 사기 검출

## 01 분류 실습 예제

Imbalanced Data (불균형 데이터)?

## 01 분류 실습 예제

Imbalanced Data (불균형 데이터)?



0.172%



99.828%

# 01 분류 실습 예제

		Predicted class	
		<i>P</i>	<i>N</i>
Actual Class	<i>P</i>	True Positives (TP)	False Negatives (FN)
	<i>N</i>	False Positives (FP)	True Negatives (TN)

$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP} \quad (\text{정확도})$$

$$Precision = \frac{TP}{TP + FP} \quad (\text{정밀도})$$

$$Recall = \frac{TP}{TP + FN} \quad (\text{재현율})$$

## 01 분류 실습 예제


$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP} \quad (\text{정확도})$$

$$Recall = \frac{TP}{TP + FN} \quad (\text{재현율})$$

불균형 데이터셋에서는  
정확도는 매우 높지만, 재현율은 급격히 떨어지는 문제 발생

## 01 분류 실습 예제

$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP} \quad (\text{정확도}) \rightarrow 94\%$$

$$Recall = \frac{TP}{TP + FN} \quad (\text{재현율})$$


100개의 표본을 가진 데이터가 0이 94개,과 1이 6개로 이루어져 있다면?



## 01 분류 실습 예제

사기 검출(Fraud Detection)

이상 검출(Anomaly Detection)

# 01 분류 실습 예제 - 캐글 신용카드 사기 검출

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline

card_df = pd.read_csv('creditcard.csv')
card_df
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...
1	0.0	1.191857	0.286151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...
...	...	...	...	...	...	...	...	...	...	...	...
284802	172786.0	-11.881118	10.071785	-9.834783	-2.068658	-5.364473	-2.608837	-4.918215	7.305334	1.914428	...
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...

284807 rows × 31 columns

# 01 분류 실습 예제 - 캐글 신용카드 사기 검출

```
from sklearn.model_selection import train_test_split
```

*#인자로 입력받은 DataFrame을 복사한 뒤 Time 칼럼만 삭제하고 복사된 DataFrame 변환*

```
def get_preprocessed_df(df=None):  
    df_copy = df.copy()  
    df_copy.drop('Time', axis=1, inplace=True)  
    return df_copy
```

*#사전 데이터 가공 후 학습과 테스트 데이터 세트를 반환하는 함수*

```
def get_train_test_dataset(df=None):  
    df_copy = get_preprocessed_df(df)  
    X_features = df_copy.iloc[:, :-1]  
    y_target = df_copy.iloc[:, -1]  
    X_train, X_test, y_train, y_test =   
    train_test_split(X_features, y_target, test_size = 0.3, random_state=0, stratify=y_target)  
    return X_train, X_test, y_train, y_test
```

# 01 분류 실습 예제 - 캐글 신용카드 사기 검출

```
X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)
```

```
#학습 데이터 세트와 테스트 데이터 세트의 레이블 값 비율을 백분율로 환산하여 서로 비슷하게 분할되었는지 확인  
print('학습 데이터 레이블 값 비율')  
print(y_train.value_counts()/y_train.shape[0] *100)  
print('테스트 데이터 레이블 값 비율')  
print(y_test.value_counts()/y_test.shape[0] *100)
```

매우 imbalanced한 데이터

학습 데이터 레이블 값 비율

0	99.827451
1	0.172549

Name: Class, dtype: float64

테스트 데이터 레이블 값 비율

0	99.826785
1	0.173215

Name: Class, dtype: float64

# 01 분류 실습 예제 – 캐글 신용카드 사기 검출

두 개의 모델을 통해 자료 불균형 해결 여부, 이상치 제거 여부 등 다양한 경우의 결과 비교

*#3장에서 이용한 get\_clf\_eval()*

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix, f1_score, roc_auc_score
```

```
def get_clf_eval(y_test, pred=None, pred_proba=None):  
    confusion = confusion_matrix(y_test, pred)  
    accuracy = accuracy_score(y_test, pred)  
    precision = precision_score(y_test, pred)  
    recall = recall_score(y_test, pred)  
    f1 = f1_score(y_test, pred)  
    roc_auc = roc_auc_score(y_test, pred_proba)  
    print('오차 행렬')  
    print(confusion)  
    print('정확도 : {0:.4f}, 정밀도 : {1:.4f}, 재현율 : {2:.4f}, F1: {3:.4f}, AUC:{4:.4f}'.format(accuracy, precision, recall, f1, roc_auc))
```

# 01 분류 실습 예제 – 캐글 신용카드 사기 검출

첫 번째 모델 : 기본 머신러닝 기법 중 하나로 자료 형태에 영향을 받는 로지스틱 회귀

*#로지스틱 회귀를 이용해 신용 카드 사기 여부 예측*

```
from sklearn.linear_model import LogisticRegression

lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train)
lr_pred = lr_clf.predict(X_test)
lr_pred_proba = lr_clf.predict_proba(X_test)[:,1]
```

*#3장에서 사용한 get\_clf\_eval() 함수를 이용하여 평가 수행*  
get\_clf\_eval(y\_test, lr\_pred, lr\_pred\_proba)

오차 행렬

```
[[85282  13]
 [   56   92]]
```

정확도 : 0.9992, 정밀도 : 0.8762, 재현율 : 0.6216, F1: 0.7273, AUC:0.9582

# 01 분류 실습 예제 - 캐글 신용카드 사기 검출

두 번째 모델 : 자료 형태에 덜 민감한 앙상블 기법 중 성능이 좋은 lightgbm

```
def get_model_train_eval(model, ftr_train=None, ftr_test=None, tgt_train=None, tgt_test=None):  
    model.fit(ftr_train, tgt_train)  
    pred = model.predict(ftr_test)  
    pred_proba = model.predict_proba(ftr_test)[:,-1]  
    get_clf_eval(tgt_test, pred, pred_proba)
```

*# LightGBM으로 모델 학습 후 별도의 테스트 데이터 세트에서 예측 평가를 수행*

```
from lightgbm import LGBMClassifier
```

```
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1, boost_from_average=False)  
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test, tgt_train=y_train, tgt_test=y_test)
```

LightGBM이 버전업되며 디폴트 값이 True로 변경되었으므로  
False로 파라미터를 설정해야 함.

오차 행렬

```
[[85290    5]  
 [    37   111]]
```

정확도 : 0.9995, 정밀도 : 0.9569, 재현율 : 0.7500, F1 : 0.8409, AUC:0.9779

# 01 분류 실습 예제 – 캐글 신용카드 사기 검출

## *Logistic regression*

오차 행렬  
[[85282 13]  
[ 56 92]]  
정확도 : 0.9992, 정밀도 : 0.8762, 재현율 : 0.6216, F1: 0.7273, AUC:0.9582

재현율과 ROC-AUC이  
로지스틱 회귀보다는 높은 수치를 나타냈으나,  
여전히 낮음  
- 불균형 데이터를 처리해야 함!

&

## *LightGBM*

오차 행렬  
[[85290 5]  
[ 36 112]]  
정확도 : 0.9995, 정밀도 : 0.9573, 재현율 : 0.7568, F1: 0.8453, AUC:0.9790



# 01 분류 실습 예제

데이터 분포도 변환 – StandardScaler

데이터 분포도 변환 – 로그 변환

이상치 데이터 제거

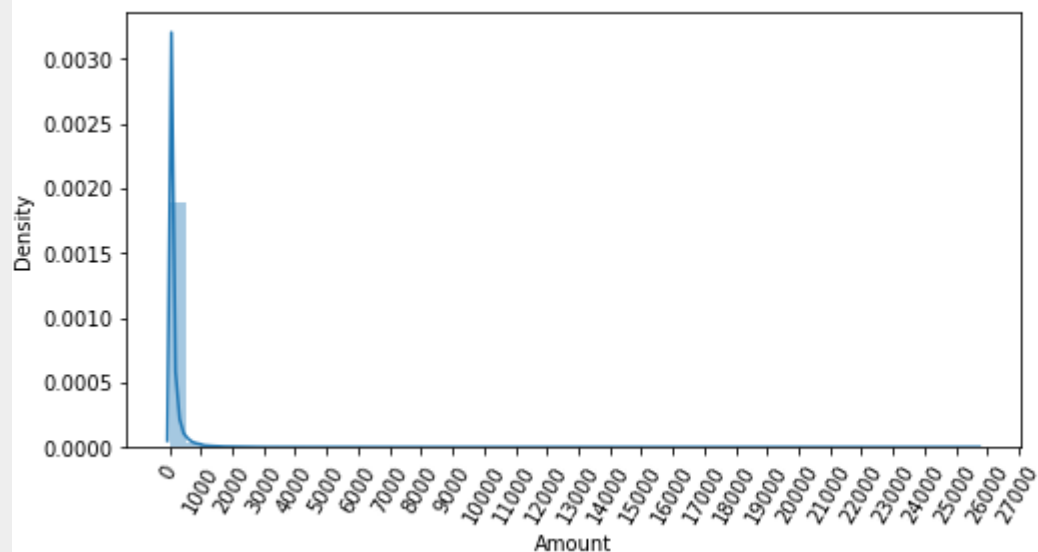
SMOTE 오버 샘플링

# 01 분류 실습 예제

Amount 피쳐 : 신용카드 사용 금액  
정상/사기 트랜잭션을 결정하는 매우 중요한 속성

```
#Amount 피쳐의 분포 확인
import seaborn as sns
plt.figure(figsize=(8,4))
plt.xticks(range(0,30000,1000), rotation=60)
sns.distplot(card_df['Amount'])
```

<AxesSubplot:xlabel='Amount', ylabel='Density'>



StandardScaler

# 01 분류 실습 예제

## StandardScaler

StandardScaler를 통해 Amount 피처를 정규 분포 형태로 변환

```
#StandardScaler으로 Amount 피처를 정규 분포 형태로 반환
from sklearn.preprocessing import StandardScaler

#위에서 정의했던 get_preprocessed_df()를 정규분포 형태로 피처값 반환하는 로직으로 수정
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    scaler = StandardScaler()
    amount_n = scaler.fit_transform(df_copy['Amount'].values.reshape(-1,1))
    #변환된 Amount를 Amount_Scaled으로 피처명 변경 후 DataFrame만 앞 칼럼으로 입력
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    #기존 Time, Amount 피처 삭제
    df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)
    return df_copy
```

# 01 분류 실습 예제

## StandardScaler

*#Amount를 정규 분포 형태로 변환 후 로지스틱 회귀 및 LightGBM 수행*

```
X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)
```

```
print('### 로지스틱 회귀 예측 성능 ###')
```

```
lr_clf = LogisticRegression()
```

```
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test,  
                     tgt_train=y_train, tgt_test=y_test)
```

```
print('### LightGBM 예측 성능 ###')
```

```
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1, boost_from_average=False)
```

```
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test,  
                     tgt_train=y_train, tgt_test=y_test)
```

### 로지스틱 회귀 예측 성능 ###

오차 행렬

```
[[85281    14]  
 [    58   90]]
```

정확도 : 0.9992, 정밀도 : 0.8654, 재현율 : 0.6081, F1: 0.7143, AUC:0.9702

### LightGBM 예측 성능 ###

오차 행렬

```
[[85290     5]  
 [    37  111]]
```

정확도 : 0.9995, 정밀도 : 0.9569, 재현율 : 0.7500, F1: 0.8409, AUC:0.9779

# 01 분류 실습 예제

데이터 분포가 심하게 왜곡되어 있을 경우 적용

원래 값을 로그값으로 변환하여 큰 값을 상대적으로 작은 값으로 변환하고, 데이터 분포도의 왜곡 개선

로그 변환

#로그 변환은 데이터 분포도가 심히 왜곡되어 있을 경우 적용하는 중요 기법 중 하나

```
def get_preprocessed_df(df=None):  
    df_copy = df.copy()  
    #numpy의 log1p()를 이용해 Amount를 로그 변환  
    amount_n = np.log1p(df_copy['Amount'])  
    df_copy.insert(0, 'Amount_Scaled', amount_n)  
    #기존 Time, Amount 피처 삭제  
    df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)  
    return df_copy
```

# 01 분류 실습 예제

## 로그 변환

```
X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)

print('### 로지스틱 회귀 예측 성능 ###')
lr_clf = LogisticRegression()
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test,
                    tgt_train=y_train, tgt_test=y_test)

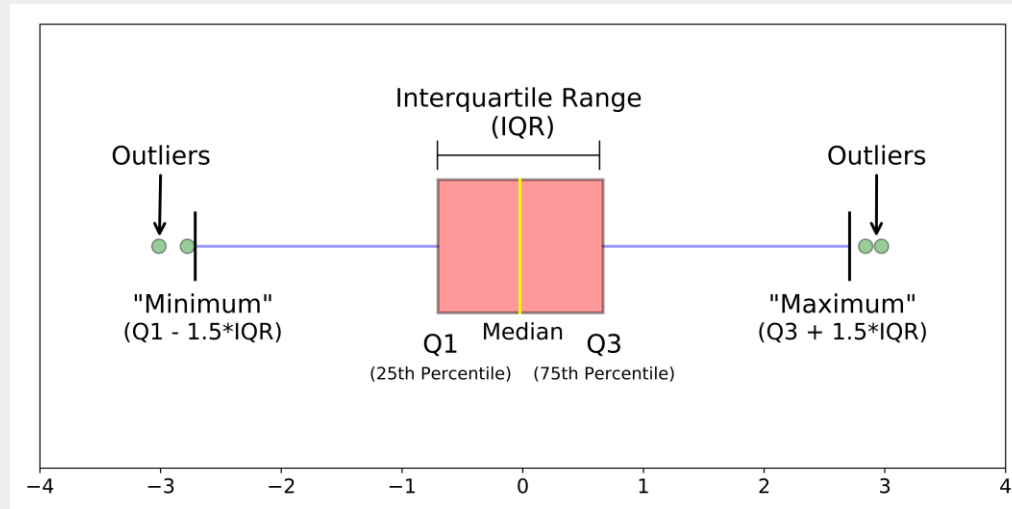
print('### LightGBM 예측 성능 ###')
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1, boost_from_average=False)
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test,
                    tgt_train=y_train, tgt_test=y_test)
```

### 로지스틱 회귀 예측 성능 ###  
오차 행렬  
[[85283 12]  
 [ 59 89]]  
정확도 : 0.9992, 정밀도 : 0.8812, 재현율 : 0.6014, F1: 0.7149, AUC:0.9727

### LightGBM 예측 성능 ###  
오차 행렬  
[[85290 5]  
 [ 35 113]]  
정확도 : 0.9995, 정밀도 : 0.9576, 재현율 : 0.7635, F1: 0.8496, AUC:0.9796

# 01 분류 실습 예제

이상치 데이터 제거



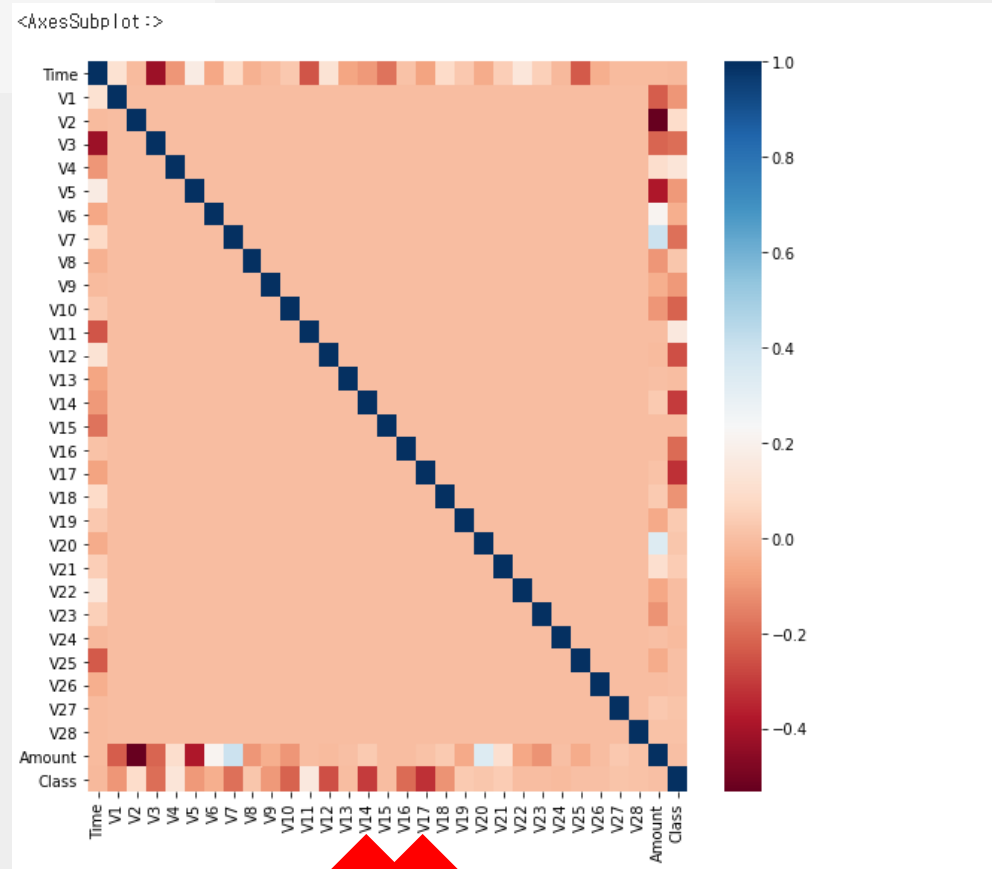
Box plot

# 01 분류 실습 예제

```
import seaborn as sns
```

```
#DataFrame의 corr()를 이용하여 각 피처별 상관도를 구한 뒤 시각화  
plt.figure(figsize=(9,9))  
corr = card_df.corr()  
sns.heatmap(corr, cmap='RdBu')
```

이상치 데이터 제거



V14, V17이 class 피처와 음의 상관관계가 높음



# 01 분류 실습 예제

## 이상치 데이터 제거

#IQR을 이용해 이상치를 검출하는 함수를 생성한뒤, 이를 이용해 검출된 이상치 삭제

```
import numpy as np

def get_outlier(df=None, column=None, weight=1.5):
    #fraud에 해당하는 column 데이터만 추출, 1/4분위와 3/4분위 지점을 np.percentile로 구함
    fraud = df[df['Class']==1][column]
    quantile_25=np.percentile(fraud.values,25)
    quantile_75=np.percentile(fraud.values,75)
    #IQR을 구하고, IQR에 1.5를 곱해 최댓값과 최솟값 지점 구함
    iqr = quantile_75-quantile_25
    iqr_weight = iqr*weight
    lowest_val = quantile_25 - iqr_weight
    highest_val = quantile_75 + iqr_weight
    #최댓값보다 크거나 최솟값보다 작은 값을 이상치 데이터로 설정하고 DataFrame index 반환
    outlier_index = fraud[(fraud<lowest_val) | (fraud>highest_val)].index
    return outlier_index
```

```
outlier_index = get_outlier(df=card_df, column='V14', weight=1.5)
print('이상치 데이터 인덱스:', outlier_index)
```

```
이상치 데이터 인덱스: Int64Index([8296, 8615, 9035, 9252], dtype='int64')
```

# 01 분류 실습 예제

이상치 데이터 제거

*#get\_outlier()를 이용해 이상치를 추출하고 이를 삭제하는 로직을 get\_preprocessed\_df()함수에 추가*

```
def get_preprocessed_df(df=None):
    df_copy = df.copy()
    amount_n = np.log1p(df_copy['Amount'])
    df_copy.insert(0, 'Amount_Scaled', amount_n)
    df_copy.drop(['Time', 'Amount'], axis=1, inplace=True)
    #이상치 데이터 삭제하는 로직 추가
    outlier_index=get_outlier(df=df_copy,column='V14',weight=1.5)
    df_copy.drop(outlier_index,axis=0, inplace=True)
    return df_copy
```

# 01 분류 실습 예제

이상치 데이터 제거

```
X_train, X_test, y_train, y_test = get_train_test_dataset(card_df)

print('### 로지스틱 회귀 예측 성능 ###')
lr_clf = LogisticRegression()
get_model_train_eval(lr_clf, ftr_train=X_train, ftr_test=X_test,
                    tgt_train=y_train, tgt_test=y_test)

print('### LightGBM 예측 성능 ###')
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1, boost_from_average=False)
get_model_train_eval(lgbm_clf, ftr_train=X_train, ftr_test=X_test,
                    tgt_train=y_train, tgt_test=y_test)

### 로지스틱 회귀 예측 성능 ###
오차 행렬
[[85281    14]
 [    48    98]]
정확도 : 0.9993, 정밀도 : 0.8750, 재현율 : 0.6712, F1: 0.7597, AUC:0.9743
### LightGBM 예측 성능 ###
오차 행렬
[[85290     5]
 [    25   121]]
정확도 : 0.9996, 정밀도 : 0.9603, 재현율 : 0.8288, F1: 0.8897, AUC:0.9780
```

## 01 분류 실습 예제

# SMOTE 사용 전!

**pip** : `pip install imbalanced-learn`

**anaconda** : `conda install -c conda-forge imbalanced-learn`



SMOTE

# 01 분류 실습 예제

```
#SMOTE를 적용할 때는 항상 학습 데이터셋만 오버샘플링 해야함
#SMOTE 객체의 fit_sample() 메소드를 이용해 증식한 뒤 데이터를 증식 전과 비교
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=0)
X_train_over, y_train_over = smote.fit_sample(X_train, y_train)
print('SMOTE 적용 전 학습용 피쳐/레이블 데이터 세트: ', X_train.shape, y_train.shape)
print('SMOTE 적용 후 학습용 피쳐/레이블 데이터 세트: ', X_train_over.shape, y_train_over.shape)
print('SMOTE 적용 후 레이블값 분포: \n', pd.Series(y_train_over).value_counts())
```

```
SMOTE 적용 전 학습용 피쳐/레이블 데이터 세트: (199362, 29) (199362,)
SMOTE 적용 후 학습용 피쳐/레이블 데이터 세트: (398040, 29) (398040,)
SMOTE 적용 후 레이블값 분포:
1    199020
0    199020
Name: Class, dtype: int64
```

로지스틱 회귀 모델이 오버 샘플링으로 인해 실제 원본 데이터의 유형보다 더  
무나 많은 Class가 1인 데이터를 학습하여 실제 테스트 데이터에서 예측을 지  
나치게 Class 1로 적용해 정밀도가 급격하게 떨어지는 것

```
#학습 데이터 세트 기반으로 로지스틱 회귀 모델 학습
lr_clf = LogisticRegression()
#ftr_train과 tgt_train 인자 값이 SMOTE 증식된 X_train_over과 y_train_over으로 변경될때 유의
get_model_train_eval(lr_clf, ftr_train=X_train_over, ftr_test=X_test,
                    tgt_train=y_train_over, tgt_test=y_test)
```

```
오차 행렬
[[82937 2358]
 [  11 135]]
정확도 : 0.9723, 정밀도 : 0.0542, 재현율 : 0.9247, F1: 0.1023, AUC:0.9737
```

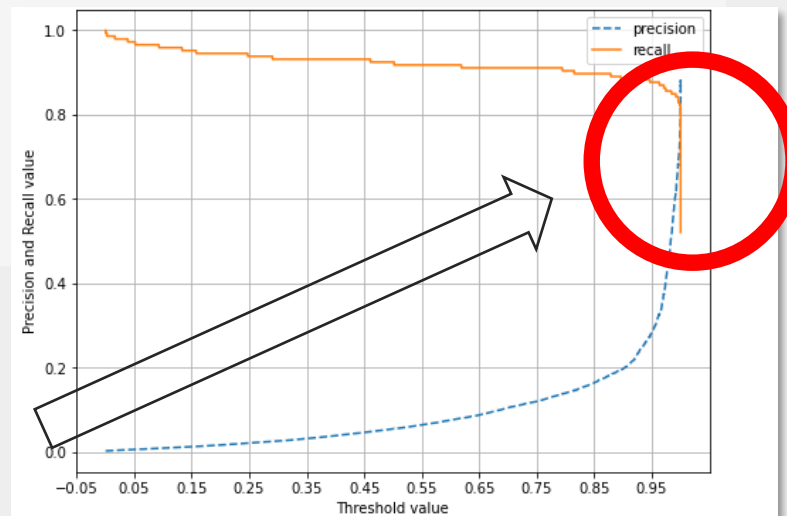
SMOTE

# 01 분류 실습 예제

```
#정밀도가 극도로 저하되므로 현실 업무에 사용 불가.  
#로지스틱 회귀 모델이 오버 샘플링으로 인해 실제 원본 데이터의 유형보다 너무나 많은 Class=1 데이터 학습해서 정밀도가 급격히 떨어짐.  
#정밀도에 어떠한 문제가 생겼는지 시각적으로 확인  
from sklearn.metrics import precision_recall_curve  
import matplotlib.pyplot as plt  
import matplotlib.ticker as ticker  
%matplotlib inline  
  
def precision_recall_curve_plot(y_test, pred_proba_c1):  
    precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_c1)  
  
    plt.figure(figsize=(8,6))  
    threshold_boundary = thresholds.shape[0]  
    plt.plot(thresholds, precisions[0:threshold_boundary], linestyle='--', label='precision')  
    plt.plot(thresholds, recalls[0:threshold_boundary], label='recall')  
  
    start, end = plt.xlim()  
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))  
  
    plt.xlabel('Threshold value'); plt.ylabel('Precision and Recall value')  
    plt.legend(); plt.grid()  
    plt.show()  
    #3장
```

SMOTE

임계값 0.99 이하에서는 재현율이 매우 높고 정밀도가 극단적으로 낮고,  
이상에서는 재현율이 극단적으로 낮아지고 정밀도가 급격히 상승



# 01 분류 실습 예제

```
lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1, boost_from_average=False)
get_model_train_eval(lgbm_clf, ftr_train=X_train_over, ftr_test=X_test,
                    tgt_train=y_train_over, tgt_test=y_test)
```

*#SMOTE 적용 시 재현율은 높아지나 정밀도는 낮아지는 것이 일반적*

오차 행렬

```
[[85283   12]
 [   22  124]]
```

정확도 : 0.9996, 정밀도 : 0.9118, 재현율 : 0.8493, F1: 0.8794, AUC:0.9814

SMOTE

# 01 분류 실습 예제

	머신러닝 알고리즘	평가지표		
		정밀도	재현율	ROC-AUC
데이터 가공 없음	로지스틱 회귀	0.8762	0.6216	0.9582
	LightGBM	0.9569	0.7500	0.9779
StandardScaler	로지스틱 회귀	0.8654	0.6081	0.9702
	LightGBM	0.9569	0.7500	0.9779
데이터 로그 변환	로지스틱 회귀	0.8812	0.6041	0.9727
	LightGBM	0.9576	0.7635	0.9796
이상치 데이터 제거	로지스틱 회귀	0.8750	0.6712	0.9743
	LightGBM	0.9603	0.8288	0.9780
SMOTE 오버 샘플링	로지스틱 회귀	0.0542	0.9247	0.9737
	LightGBM	0.9118	0.8493	0.9814



## 02 분류모델 실습

---

Mobile Phone Price Classification

## 02 분류 모델 실습 – Mobile Phone Price Classification

# 스마트폰 제조 스타트업을 시작한 승길이는 삼성, 애플과 같은 IT기업이 되기 위해 열심히 노력하고 있습니다.  
 하지만 자신이 만든 스마트폰에 어떻게 가격을 책정해야 하는지 몰라 타기업들의 데이터를 가지고 머신러닝 기술자들 모임인 비타민에 도움을 요청하였습니다.  
 우리 모두 승길이의 사업을 도와줍시다!

```
import numpy as np
import pandas as pd
```

```
dataset = pd.read_csv('mobile_train.csv')
dataset.head()
```

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	...	px_height	px_width	ram	sc_h	sc_w	talk_time	three_g	touch_screen	wifi	price_range
0	842	0	2.2	0	1	0	7	0.6	188	2	...	20	756	2549	9	7	19	0	0	1	1
1	1021	1	0.5	1	0	1	53	0.7	136	3	...	905	1988	2631	17	3	7	1	1	0	2
2	563	1	0.5	1	2	1	41	0.9	145	5	...	1263	1716	2603	11	2	9	1	1	0	2
3	615	1	2.5	0	0	0	10	0.8	131	6	...	1216	1786	2769	16	8	11	1	0	0	2
4	1821	1	1.2	0	13	1	44	0.6	141	2	...	1208	1212	1411	8	2	15	1	1	0	1

5 rows × 21 columns

```
dataset['price_range'].unique()
```

```
array([1, 2, 3, 0])
```

# 총 2000개의 row로 구성되어 있고 스마트폰의 스펙을 구성하는 feature는 20개입니다.  
 # 분류해야 할 target인 'price range'는 0-3이고, 어느 쪽이 비싼 가격인지는 분석 과정에서 알아보도록 하겠습니다.

## 02 분류 모델 실습 – Mobile Phone Price Classification

스마트폰에 탑재된 배터리 용량  
블루투스 보유 여부  
AP의 명령 실행 속도  
듀얼 심 지원 여부  
전면 카메라 메가 픽셀  
4G 지원 여부  
내장 메모리  
스마트폰 깊이  
스마트폰 무게  
AP의 코어 수  
기본 카메라 메가 픽셀  
픽셀 해상도 높이  
픽셀 해상도 너비  
램 용량  
스마트폰 화면 높이  
스마트폰 화면 너비  
단일 배터리의 최대 지속 기간  
3G 지원 여부  
터치 스크린 지원 여부  
Wifi 지원 여부  
스마트폰 가격 범위

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 21 columns):
#   Column              Non-Null Count  Dtype
---  -
0   battery_power       2000 non-null   int64
1   blue                 2000 non-null   int64
2   clock_speed         2000 non-null   float64
3   dual_sim            2000 non-null   int64
4   fc                   2000 non-null   int64
5   four_g              2000 non-null   int64
6   int_memory          2000 non-null   int64
7   m_dep               2000 non-null   float64
8   mobile_wt           2000 non-null   int64
9   n_cores             2000 non-null   int64
10  pc                   2000 non-null   int64
11  px_height            2000 non-null   int64
12  px_width             2000 non-null   int64
13  ram                  2000 non-null   int64
14  sc_h                 2000 non-null   int64
15  sc_w                 2000 non-null   int64
16  talk_time            2000 non-null   int64
17  three_g              2000 non-null   int64
18  touch_screen         2000 non-null   int64
19  wifi                 2000 non-null   int64
20  price_range          2000 non-null   int64
dtypes: float64(2), int64(19)
memory usage: 328.2 KB
```

# 결측치가 있으면 당연히 모델에 적용할 수 없겠죠?  
결측치를 확인해봅니다.

# 다행히 결측치는 없네요!

# 모두 Encoding이 되어 있어 이에 대한  
추가적인 전처리는 필요 없어보입니다!

## 02 분류 모델 실습 – Mobile Phone Price Classification

# binary 값을 가진 feature는 다음과 같습니다.

- Blue : bluetooth가 있으면 1, 없으면 0
- Dual sim : dual sim이면 1, 아니면 0
- Four\_g : 4G를 지원하면 1, 아니면 0
- Three\_g : 3G를 지원하면 1, 아니면 0
- Touch screen : 터치스크린을 지원하면 1, 아니면 0
- Wifi : 와이파이를 지원하면 1, 아니면 0

# binary가 아닌 feature는 각각 넓은 범위의 값을 가지고 있는 것 같습니다.

-> **scaling 필요**

```
dataset.describe().T
```

	count	mean	std	min	25%	50%	75%	max
battery_power	2000.0	1238.51850	439.418206	501.0	851.75	1226.0	1615.25	1998.0
blue	2000.0	0.49500	0.500100	0.0	0.00	0.0	1.00	1.0
clock_speed	2000.0	1.52225	0.816004	0.5	0.70	1.5	2.20	3.0
dual_sim	2000.0	0.50950	0.500035	0.0	0.00	1.0	1.00	1.0
fc	2000.0	4.30950	4.341444	0.0	1.00	3.0	7.00	19.0
four_g	2000.0	0.52150	0.499662	0.0	0.00	1.0	1.00	1.0
int_memory	2000.0	32.04650	18.145715	2.0	16.00	32.0	48.00	64.0
m_dep	2000.0	0.50175	0.288416	0.1	0.20	0.5	0.80	1.0
mobile_wt	2000.0	140.24900	35.399655	80.0	109.00	141.0	170.00	200.0
n_cores	2000.0	4.52050	2.287837	1.0	3.00	4.0	7.00	8.0
pc	2000.0	9.91650	6.064315	0.0	5.00	10.0	15.00	20.0
px_height	2000.0	645.10800	443.780811	0.0	282.75	564.0	947.25	1960.0
px_width	2000.0	1251.51550	432.199447	500.0	874.75	1247.0	1633.00	1998.0
ram	2000.0	2124.21300	1084.732044	256.0	1207.50	2146.5	3064.50	3998.0
sc_h	2000.0	12.30650	4.213245	5.0	9.00	12.0	16.00	19.0
sc_w	2000.0	5.76700	4.356398	0.0	2.00	5.0	9.00	18.0
talk_time	2000.0	11.01100	5.463955	2.0	6.00	11.0	16.00	20.0
three_g	2000.0	0.76150	0.426273	0.0	1.00	1.0	1.00	1.0
touch_screen	2000.0	0.50300	0.500116	0.0	0.00	1.0	1.00	1.0
wifi	2000.0	0.50700	0.500076	0.0	0.00	1.0	1.00	1.0
price_range	2000.0	1.50000	1.118314	0.0	0.75	1.5	2.25	3.0

## 02 분류 모델 실습 – Mobile Phone Price Classification

# binary 값을 가진 feature 들이 어떤 비율로 데이터셋에 포함되어 있는지 알아보기 위해 가장 직관적인 pie chart로 그 분포를 알아보겠습니다.

```
import matplotlib.pyplot as plt
import seaborn as sns
import warnings

warnings.filterwarnings('ignore')
```

```
feature = ['blue', 'dual_sim', 'four_g', 'three_g', 'touch_screen', 'wifi',
           'price_range']

plt.figure(figsize = (20,10))
count = 0

for i in feature:

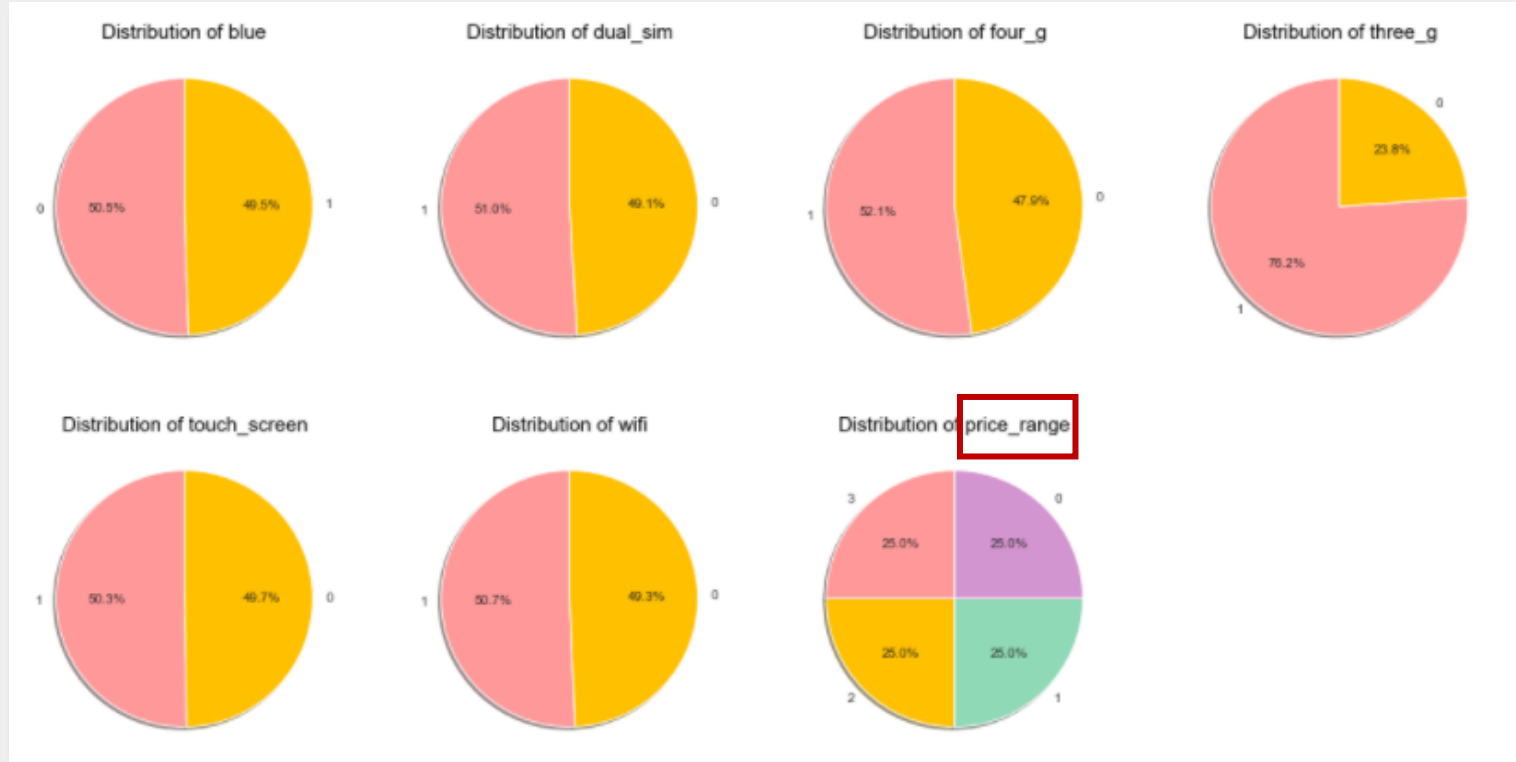
    colors = ['#ff9999', '#ffc000', '#8fd9b6', '#d395d0']
    labels = dataset[i].value_counts().index
    sizes = dataset[i].value_counts().values

    plt.subplot(2,4,count+1)
    plt.pie(sizes, labels=labels, shadow = True,
            startangle=90, autopct='%1.1f%%', colors = colors)
    plt.title(f'Distribution of {feature[count]}', color = 'black', fontsize = 15)
    count += 1
```

3주차 시각화 참고 !

## 02 분류 모델 실습 – Mobile Phone Price Classification

# binary 값을 가진 feature 들이 데이터셋에서 차지하는 비율



# 3G의 경우 75% 이상 지원을 하고 나머지의 경우는 절반씩 지원하는 것 같습니다.

# 타겟 컬럼인 'price\_range'는 각 클래스가 균일한 분포를 가지고 있어 smote는 생각하지 않아도 될 것으로 보입니다.

## 02 분류 모델 실습 – Mobile Phone Price Classification

```
In [90]: var1 = ['battery_power', 'clock_speed', 'fc', 'int_memory',
                'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_height',
                'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'price_range']

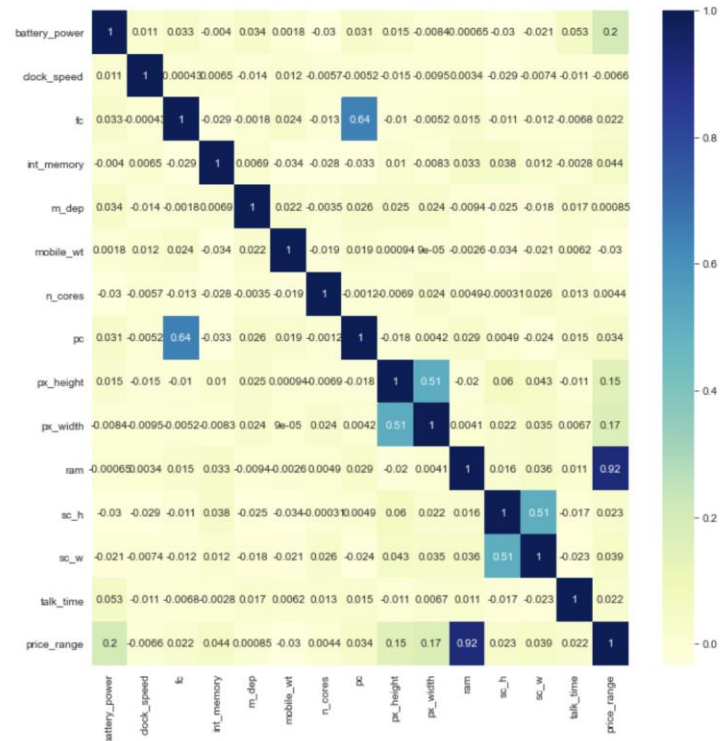
dataset_continuous_var = dataset[var1]
```

# 수치형 feature들이 어떤 **상관관계**를 보이는지  
Heatmap으로 표현해 보았습니다.

# 눈에 띄는 상관관계는 4군데 입니다.

- 전면카메라 스펙(fc)과 후면카메라 스펙(pc)
- 스크린의 가로크기(sc\_w)와 세로크기(sc\_h)
- 디스플레이 해상도 너비(px\_width)와 높이(px\_height)
- 가격(price\_range)와 랩(ram)

```
In [91]: plt.figure(figsize = (11,11))
sns.heatmap(dataset_continuous_var.corr(),annot=True,cmap='YlGnBu')
plt.show()
```



## 02 분류 모델 실습 – Mobile Phone Price Classification

# Ram과 Price\_range의 관계

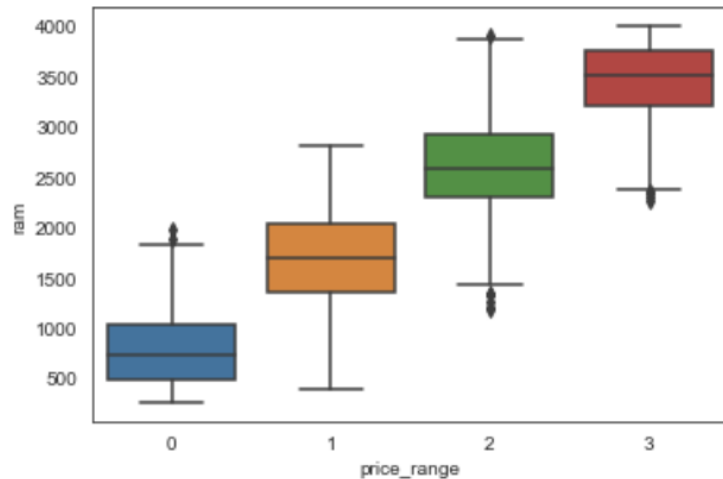
# 히트맵을 통해 Ram이 'price\_range'에 가장 큰 영향을 주는 것을 알 수 있었습니다.

# 그렇다면 ram과 price range의 plot을 그려 price range의 어떤 값이 높은 값인지 알 수 있을 것 같네요!

# 이를 Box plot으로 그리면 다음과 같이 나옵니다.

```
In [25]: sns.boxplot(x='price_range', y='ram', data=dataset)
```

```
Out[25]: <AxesSubplot:xlabel='price_range', ylabel='ram'>
```



# ram 용량수로 미루어 보아  
'price\_range'는 0이 낮은 가격대, 3이 높은  
가격대인 것을 알 수 있습니다. (다다익람!)



# 02 분류 모델 실습 – Mobile Phone Price Classification

# 본격적으로 분류 실습을 하기 전, 전처리를 해줍니다.

```
# X에는 feature들만, y에는 컬럼 'price_range'를 넣어준다.
```

```
X = dataset.iloc[:, :-1]
```

```
y = dataset.iloc[:, -1]
```

```
# feature들의 값이 서로 다른 범위이므로 scaling을 통해 같은 범위로 조정해주어야한다.
```

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
```

```
var = ['battery_power', 'clock_speed', 'fc', 'int_memory',  
       'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_height',  
       'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time']
```

```
X_scaled = scaler.fit_transform(X[var])
```

```
X_scaled_df = pd.DataFrame(X_scaled, columns=var)
```

```
for i in var:
```

```
    X[i] = X_scaled_df[i] # 정규화된 피쳐들을 원본 데이터에 넣어준다.
```

# feature와 target을 x와 y에 할당해줍니다.

# binary가 아닌 값들은 scaling을 해줍니다.

## 02 분류 모델 실습 – Mobile Phone Price Classification

### # Preprocessing

# test set size가 0.2가 되도록 train set과 test set을 분리해주었습니다.

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=101)
```

```
print(f'Total number of sample in whole dataset: {len(X)}')
```

```
print(f'Total number of sample in train dataset: {len(X_train)}')
```

```
print(f'Total number of sample in test dataset: {len(X_test)}')
```

```
Total number of sample in whole dataset: 2000
```

```
Total number of sample in train dataset: 1600
```

```
Total number of sample in test dataset: 400
```

## 02 분류 모델 실습 – Mobile Phone Price Classification

# 여러 모델로 학습을 진행한 뒤 가장 성능이 뛰어난 2개의 모델을 골라보겠습니다.

# 데이터셋이 적은 관계로 **cv 기반 학습**을 진행하겠습니다.

```
# CV (교차 검증)
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict

# 알고리즘
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Boost 관련 알고리즘
from xgboost import XGBClassifier
from xgboost import plot_importance
from lightgbm import LGBMClassifier

# 성능 평가 지표
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.metrics import precision_score, recall_score, f1_score

# GridSearchCV
from sklearn.model_selection import GridSearchCV
```

```
# 모델의 성능을 평가하는 함수 model_evaluation
def model_evaluation(y_test, pred):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred, average = 'macro')
    recall = recall_score(y_test, pred, average = 'macro')
    f1 = f1_score(y_test, pred, average = 'macro')

    print(confusion)
    print("Accuracy: ", accuracy)
    print("Precision: ", precision)
    print("Recall: ", recall)
    print("F1: ", f1)
```

# 필요한 라이브러리들을 import 합니다.

# 학습한 모델의 오차 행렬, 정확도, 정밀도, 재현율과 f1 score 까지 확인해보겠습니다.

## 02 분류 모델 실습 – Mobile Phone Price Classification

```
from sklearn.model_selection import StratifiedKFold
cv_results_acc = [] # cv 기반 정확도 평균, 아래 시각화에서 쓰입니다!
```

```
models = {
    'RandomForestClassifier': RandomForestClassifier(),
    'SupportVectorMachine': SVC(),
    'DecisionTreeClassifier': DecisionTreeClassifier(),
    'KNeighborsClassifier': KNeighborsClassifier(),
    'XGBoostClassifier': XGBClassifier(),
    'LGBMClassifier': LGBMClassifier()
}
```

# 저희가 지금 까지 배운  
Random Forest, SVM, Decision Tree, K-neighbors, XGBoost, LightGBM 모델을  
이용해보겠습니다!

```
for m in models:
    clf = models[m]
    clf.fit(X_train, y_train) # 학습
    pred = clf.predict(X_test) # 예측

    print('\n*****', m, '*****')
```

```
    model_evaluation(y_test, pred) # 성능 평가 함수 호출
```

# 위에서 정의한 model\_evaluation()을 호출하여 성능 평가를 진행합니다.  
각 모델의 오차 행렬, 정확도, 정밀도, 재현율과 f1 score 가 출력되지요?

```
# CV 기반 (k-fold) 정확도
```

```
cv_score = cross_val_score(clf, X_train, y_train, scoring='accuracy', cv = 10)
cv_results_acc.append(cv_score.mean() * 100)
print('\nCross Validation Accuracy: {:.4f}'.format(cv_score.mean()))
```

# cross validation을 진행한 후 정확도를 도출합니다.  
# 분류에는 Stratified K-Fold, 회귀에는 K-Fold가 쓰입니다.

## 02 분류 모델 실습 – Mobile Phone Price Classification

\*\*\*\*\* RandomForestClassifier \*\*\*\*\*

```
[[87 7 0 0]
 [ 9 84 9 0]
 [ 0 16 98 6]
 [ 0 0 5 79]]
```

Accuracy: 0.8700

Precision: 0.8739

Recall: 0.8766

F1: 0.8748

Cross Validation Accuracy: 0.8806

\*\*\*\*\* SupportVectorMachine \*\*\*\*\*

```
[[ 88 6 0 0]
 [ 6 88 8 0]
 [ 0 12 101 7]
 [ 0 0 5 79]]
```

Accuracy: 0.8900

Precision: 0.8927

Recall: 0.8953

F1: 0.8937

Cross Validation Accuracy: 0.8875

\*\*\*\*\* DecisionTreeClassifier \*\*\*\*\*

```
[[80 14 0 0]
 [ 7 85 10 0]
 [ 0 21 85 14]
 [ 0 0 6 78]]
```

Accuracy: 0.8200

Precision: 0.8293

Recall: 0.8303

F1: 0.8263

Cross Validation Accuracy: 0.8369

\*\*\*\*\* KNeighborsClassifier \*\*\*\*\*

```
[[68 26 0 0]
 [27 52 20 3]
 [ 1 48 58 13]
 [ 0 3 23 58]]
```

Accuracy: 0.5900

Precision: 0.6174

Recall: 0.6018

F1: 0.6063

Cross Validation Accuracy: 0.5725

\*\*\*\*\* XGBoostClassifier \*\*\*\*\*

```
[[ 85 9 0 0]
 [ 5 92 5 0]
 [ 0 11 100 9]
 [ 0 0 5 79]]
```

Accuracy: 0.8900

Precision: 0.8932

Recall: 0.8950

F1: 0.8930

Cross Validation Accuracy: 0.9062

\*\*\*\*\* LGBMClassifier \*\*\*\*\*

```
[[ 89 5 0 0]
 [ 5 92 5 0]
 [ 0 6 105 9]
 [ 0 0 4 80]]
```

Accuracy: 0.9150

Precision: 0.9150

Recall: 0.9190

F1: 0.9167

Cross Validation Accuracy: 0.9031

## 02 분류 모델 실습 – Mobile Phone Price Classification

# 학습한 모델의 개수가 많은 만큼 한 눈에 각 모델의 성능을 확인할 수 있도록 시각화를 해보는 것이 좋을 것 같네요.

# cv score.mean을 이용하여 모델의 score들을 그래프로 그려봅시다.

```
plt.figure(figsize = (20, 10))
sns.set_style('white')

barWidth = 0.5
bars1 = cv_results_acc # cv 기반 정확도 평균값들

r1 = np.arange(len(bars1)) # cv 기반 정확도의 평균값들 갯수
r2 = [x + barWidth for x in r1] # 정확도 갯수만큼

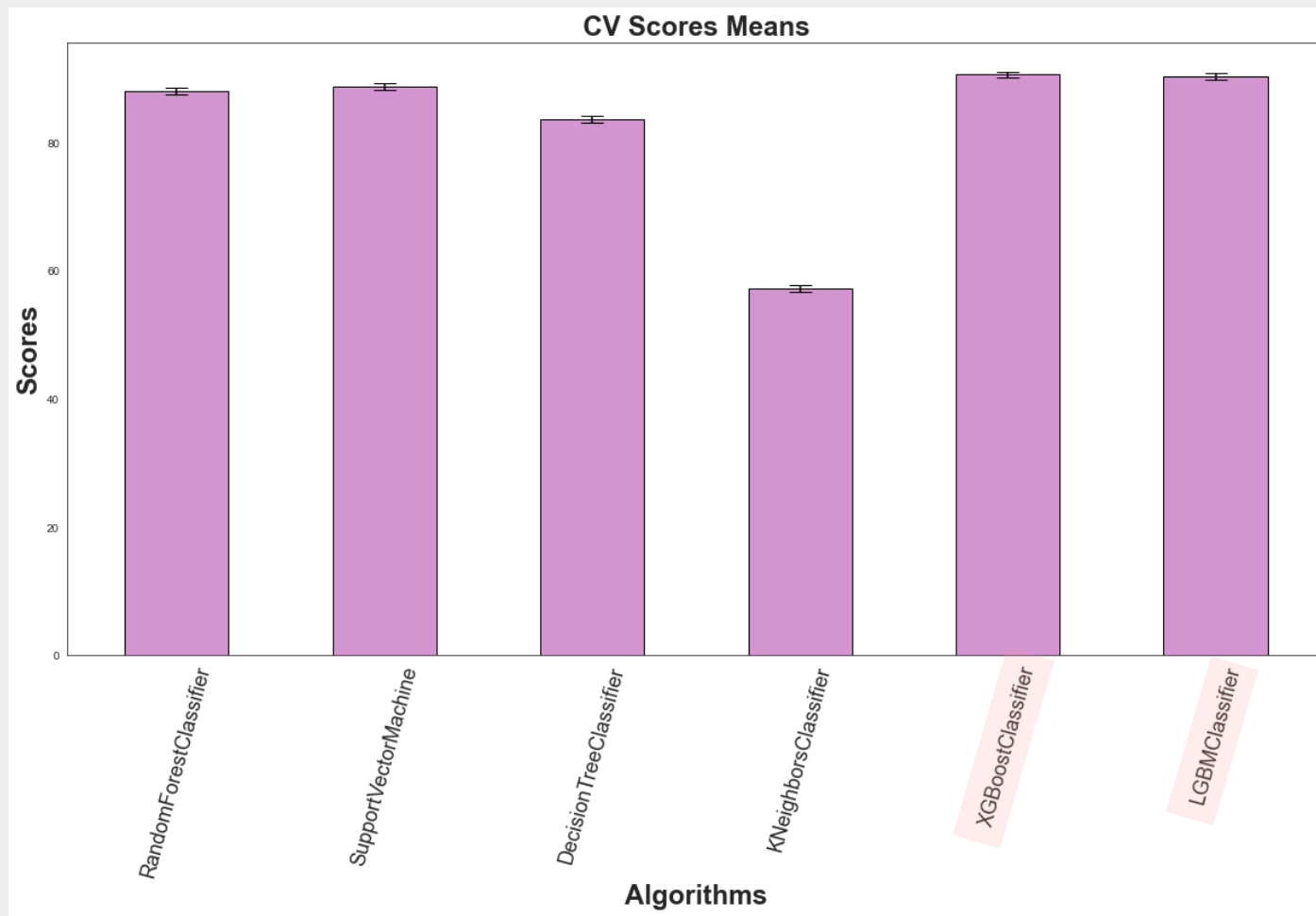
modelName = ['RandomForestClassifier', 'SupportVectorMachine',
             'DecisionTreeClassifier', 'KNeighborsClassifier',
             'XGBoostClassifier', 'LGBMClassifier']

plt.title('CV Scores Means', fontweight = 'bold', size = 24)
plt.bar(r2, bars1, color = '#d395d0', width = barWidth, edgecolor = 'black', yerr = 0.5, ecolor = 'black', capsize = 10)

plt.xlabel('Algorithms', fontweight = 'bold', size = 24)
plt.ylabel('Scores', fontweight = 'bold', size = 24)
plt.xticks([r + barWidth for r in range(len(bars1))], modelName, fontsize = 18, rotation = 75)
plt.show()
```

## 02 분류 모델 실습 – Mobile Phone Price Classification

# Modeling



# XGBoost와 LightGBM의 score가 가장 높은 것을 확인할 수 있습니다.

## 02 분류 모델 실습 – Mobile Phone Price Classification

# 정확도가 높았던 XGBoost와 LightGBM 모델의 **최적 하이퍼 파라미터**를 찾아보도록 하겠습니다.

### # XGBoost

```
xgb_clf = XGBClassifier(n_estimators = 100)

params = {'max_depth': [1, 7], 'min_child_weight': [1, 3], 'colsample_bytree': [0.5, 0.75]}

gridcv = GridSearchCV(xgb_clf, param_grid = params, cv=3)
gridcv.fit(X_train, y_train, verbose = False)
```

```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, gamma=None,
                                     gpu_id=None, importance_type='gain',
                                     interaction_constraints=None,
                                     learning_rate=None, max_delta_step=None,
                                     max_depth=None, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     n_estimators=100, n_jobs=None,
                                     num_parallel_tree=None, random_state=None,
                                     reg_alpha=None, reg_lambda=None,
                                     scale_pos_weight=None, subsample=None,
                                     tree_method=None, validate_parameters=None,
                                     verbosity=None),
             param_grid={'colsample_bytree': [0.5, 0.75], 'max_depth': [1, 7],
                         'min_child_weight': [1, 3]})
```

참고 ☺

- min\_child\_weight : child에서 필요한 모든 관측치에 대한  
가중치 합을 최소화
- colsample\_bytree : 트리 생성에 필요한 피쳐 샘플링 비율



## 02 분류 모델 실습 – Mobile Phone Price Classification

# 정확도가 높았던 XGBoost와 LightGBM 모델의 **최적 하이퍼 파라미터**를 찾아보도록 하겠습니다.

### # XGBoost

```
print('GridSearchCV 최적 파라미터: ', gridcv.best_params_)
```

```
xgb_f1_score = f1_score(y_test, gridcv.predict(X_test), average = 'macro')
```

```
print(f'F1_score: {xgb_f1_score:.4f}')
```

GridSearchCV 최적 파라미터: {'colsample\_bytree': 0.75, 'max\_depth': 7, 'min\_child\_weight': 3}

F1\_score: 0.9050

n\_estimators를 100 -> 1000으로 바꿔주고, 최적 파라미터 값을 추가합니다.

*# 위에서 나온 최적 파라미터로 설정하여 학습 후, 정확도를 확인해보자.*

```
xgb_clf = XGBClassifier(n_estimators = 1000, random_state = 156, learning_rate = 0.01, max_depth = 7,  
                        min_child_weight = 3, colsample_bytree = 0.75, reg_alpha = 0.03)
```

```
xgb_clf.fit(X_train, y_train, verbose = False) # 학습
```

```
xgb_f1_score = f1_score(y_test, xgb_clf.predict(X_test), average = 'macro')
```

```
print('F1 score: {:.4f}'.format(xgb_f1_score))
```

```
cv_score = cross_val_score(xgb_clf, X_train, y_train, scoring = 'accuracy', cv = 10)
```

```
print('Cross Validation Accuracy score: ', cv_score.mean())
```

F1\_score: 0.9050

Cross Validation Accuracy: 0.9118749999999999

## 02 분류 모델 실습 – Mobile Phone Price Classification

# 정확도가 높았던 XGBoost와 LightGBM 모델의 **최적 하이퍼 파라미터**를 찾아보도록 하겠습니다.

### # LightGBM

```
lgbm_clf = LGBMClassifier(n_estimators = 200)

params = {'num_leaves': [32, 64], 'max_depth': [128, 160],
          'min_child_samples': [60, 100], 'subsample': [0.8, 1]}

gridcv = GridSearchCV(lgbm_clf, param_grid = params, cv=3)
gridcv.fit(X_train, y_train, verbose = False)
```

```
GridSearchCV(cv=3, estimator=LGBMClassifier(n_estimators=200),
             param_grid={'max_depth': [128, 160],
                          'min_child_samples': [60, 100], 'num_leaves': [32, 64],
                          'subsample': [0.8, 1]})
```

```
print('GridSearchCV 최적 파라미터: ', gridcv.best_params_)
```

```
lgbm_f1_score = f1_score(y_test, gridcv.predict(X_test), average = 'macro')
print(f'F1_score: {lgbm_f1_score:.4f}')
```

```
GridSearchCV 최적 파라미터: {'max_depth': 128, 'min_child_samples': 100, 'num_leaves': 32, 'subsample': 0.8}
F1_score: 0.9028
```

참고 ☺

- num\_leaves : 개별 트리의 최대 리프 수
- max\_depth : 트리의 최대 깊이
- min\_child\_samples : 리프 노드가 되기 위한 최소 데이터 수
- subsample : 데이터 샘플링 비율

## 02 분류 모델 실습 – Mobile Phone Price Classification

# 정확도가 높았던 XGBoost와 LightGBM 모델의 **최적 하이퍼 파라미터**를 찾아보도록 하겠습니다.

### # LightGBM

→ n\_estimators를 200 -> 1000으로 바꿔주고, 최적 파라미터 값을 추가합니다.

```
lgbm_clf = LGBMClassifier(n_estimators = 1000, max_depth = 128,  
                           min_child_samples = 100, num_leaves = 32, subsample = 0.8)  
  
lgbm_clf.fit(X_train, y_train, verbose=False)  
  
lgbm_f1_score = f1_score(y_test, lgbm_clf.predict(X_test),  
                          average = 'macro')  
  
print(f'F1_score: {lgbm_f1_score:.4f}')  
cv_score = cross_val_score(lgbm_clf, X_train, y_train, scoring="accuracy", cv=10)  
print(f"Cross Validation Accuracy: {cv_score.mean()}")
```

```
F1_score: 0.9125  
Cross Validation Accuracy: 0.9099999999999999
```

# 승길이에게 자신이 만든 스마트폰의 스펙 데이터를 받아 LightGBM이나 XGBoost로 학습시킨 모델을 통해 승길이의 스마트폰 가격 범위를 설정해주면 되겠네요!

## 02 분류 모델 실습 – Mobile Phone Price Classification

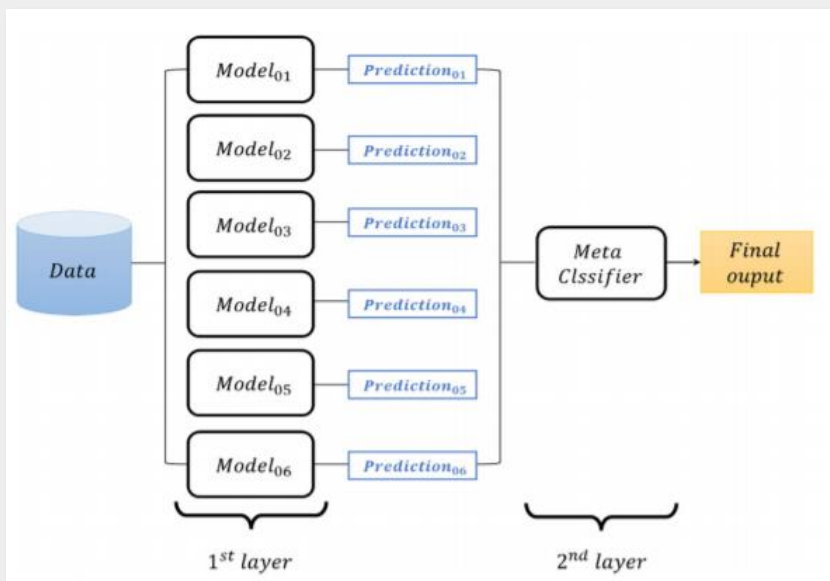
**승길이의 고민 해결!**

# 03 스택킹 앙상블

**“Two heads are better than one”**

## 03 스택킹(Stacking) 앙상블

- 여러 개의 개별 모델들이 생성한 예측 데이터를 바탕으로 최종 메타모델이 학습/예측할 데이터 세트를 재 생성하는 기법



### 배깅 및 부스팅과의 공통점

개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출한다는 점

### 배깅 및 부스팅과의 차이점

개별 알고리즘의 예측값이 새로운 학습 데이터가 되어 최종 모델이 다시 예측을 수행한다는 점

### 스태킹 앙상블 모델

- 1) 개별 모델이 원본 학습 데이터에 대한 예측 값을 스택킹 형태로 쌓는다.
- 2) 새로 형성된 데이터를 최종 모델이 다시 학습 및 예측을 수행한다.

## 03 스택킹(Stacking) 앙상블

### ○ 2가지 개념의 모델

#### 개별적인 기반 모델

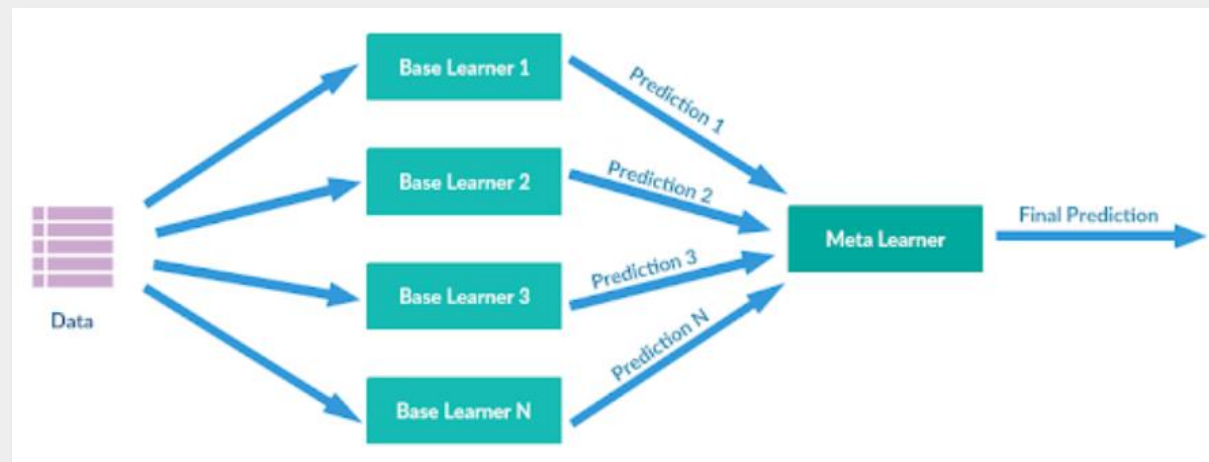
원본 학습 데이터를 학습한 후, 원본 테스트 데이터에 대해 예측하는 개별 모델

#### 최종 메타 모델

weak learner들의 예측 데이터들을 스택킹한 데이터셋을 학습한 후, 예측하는 최종 모델

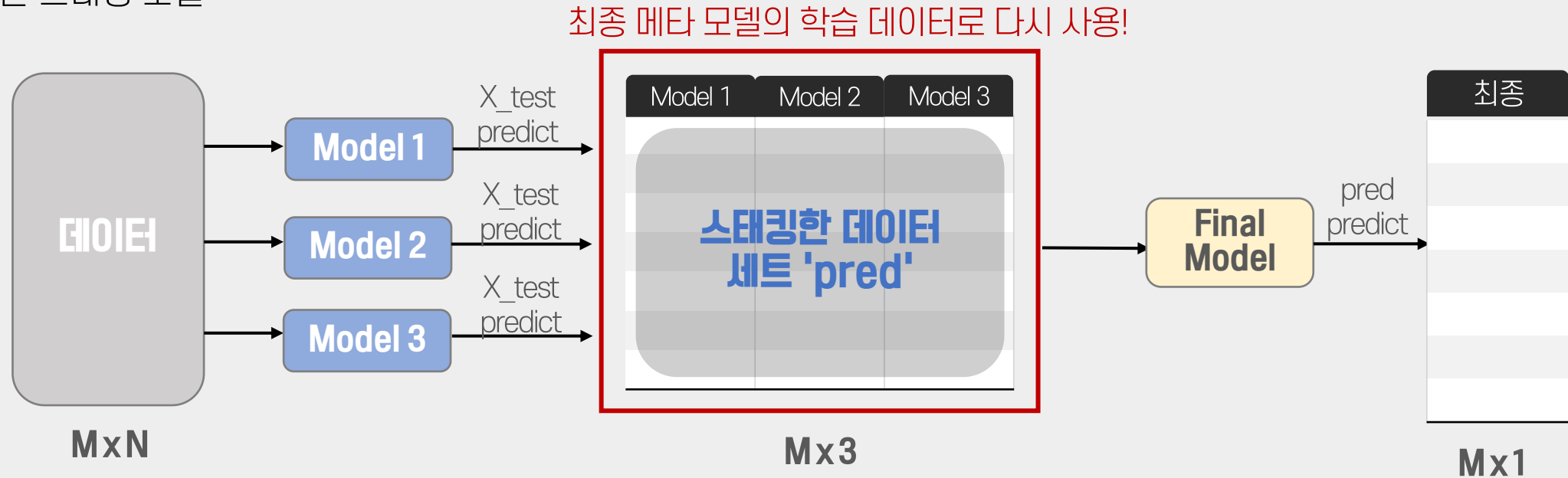
#### !! 핵심 !!

최종 메타 모델이 사용할 학습 데이터 세트와 예측 데이터 세트는 개별 모델들의 예측 값들을 스택킹 형태로 결합하여 생성된다는 점!



## 03 스택킹(Stacking) 앙상블

### ○ 기본 스택킹 모델



- M개의 로우, N개의 피처를 가진 데이터 세트 / ML 알고리즘 모델 : 3개
- **과적합의 문제 발생** ➔ 일반적으로 **Cross Validation 세트 기반의 스택킹 앙상블을 사용!**



## 03 스택킹(Stacking) 앙상블

### ○ CV 세트 기반 스택킹 모델

- 과적합 개선을 위해 개별 모델들이 각각 교차 검증으로 최종 모델을 위한 학습용/테스트용 스택킹 데이터를 생성함
- Overfitting을 피하면서, 더욱 완성도 있는 메타 모델을 완성할 수 있음

#### 2단계의 STEP

- 1) 개별 모델들은 교차 검증을 통해 학습 / 검증 / 예측을 진행함  
이때, 검증 셋에 대한 예측값과 원본 테스트 데이터에 대한 예측값을 생성함

STEP 1

➡ 메타 모델을 위한 학습용/테스트용 데이터

- 2) 개별 모델들이 생성한 데이터를 각각 스택킹 형태로 합쳐서 메타 모델이 **학습/예측할 최종 데이터 세트를 생성함**

- 3) 메타 모델은 **최종 생성된 학습 데이터 세트와 원본 학습데이터의 레이블 데이터**로 학습을 진행함

STEP 2

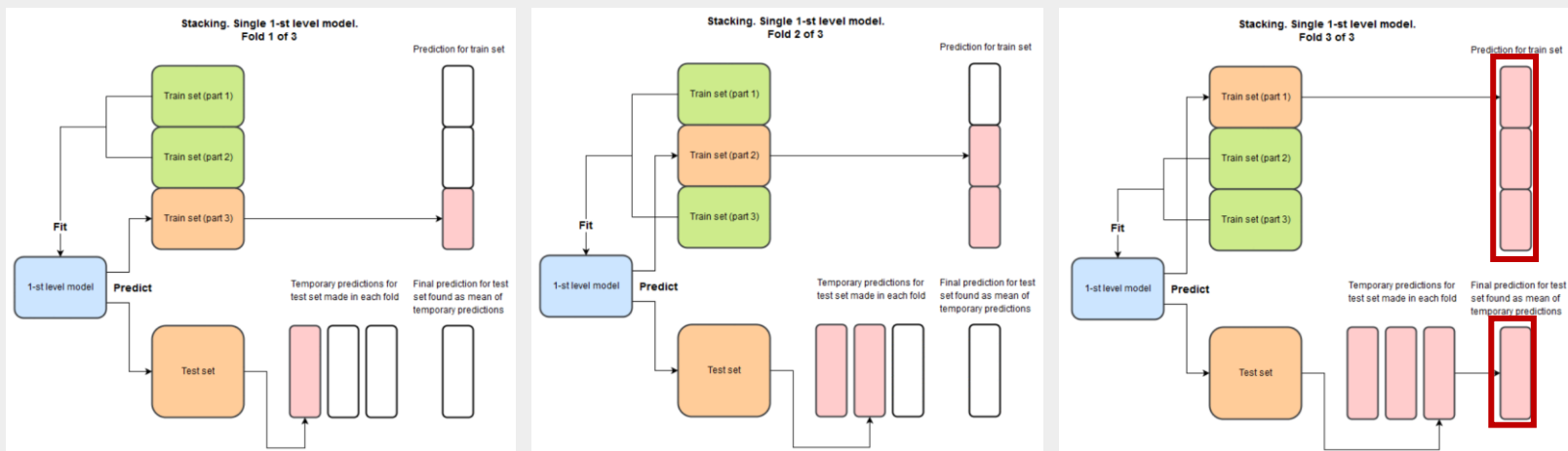
- 4) 메타 모델은 **최종 생성된 테스트 데이터 세트를 예측**하여, 원본 테스트 데이터의 레이블 데이터를 기반으로 **평가**함

# 03 스택킹(Stacking) 앙상블

## ○ CV 세트 기반 스택킹 모델

### STEP 1

기본 모델 한 개의 교차 검증을 통한  
학습/검증/예측 모습입니다!  
이러한 로직을 기본 모델들이 각자 진행합니다.



1. 원본 Train set을 N개의 fold로 나눈다. (3개의 fold로 나누었다 가정)
2. 2개의 fold를 학습을 위한 데이터 폴드로, 1개의 fold를 검증을 위한 데이터 폴드로 사용
3. 위의 2개의 폴드 데이터를 이용해 개별 모델을 학습, 1개의 폴드 데이터로 **예측 후 결과 저장**
4. 2개의 학습 폴드 데이터로 학습한 개별 모델이 원본 Test set에 대해 **예측 후 결과 저장**

교차 검증 과정 기억나시죠?

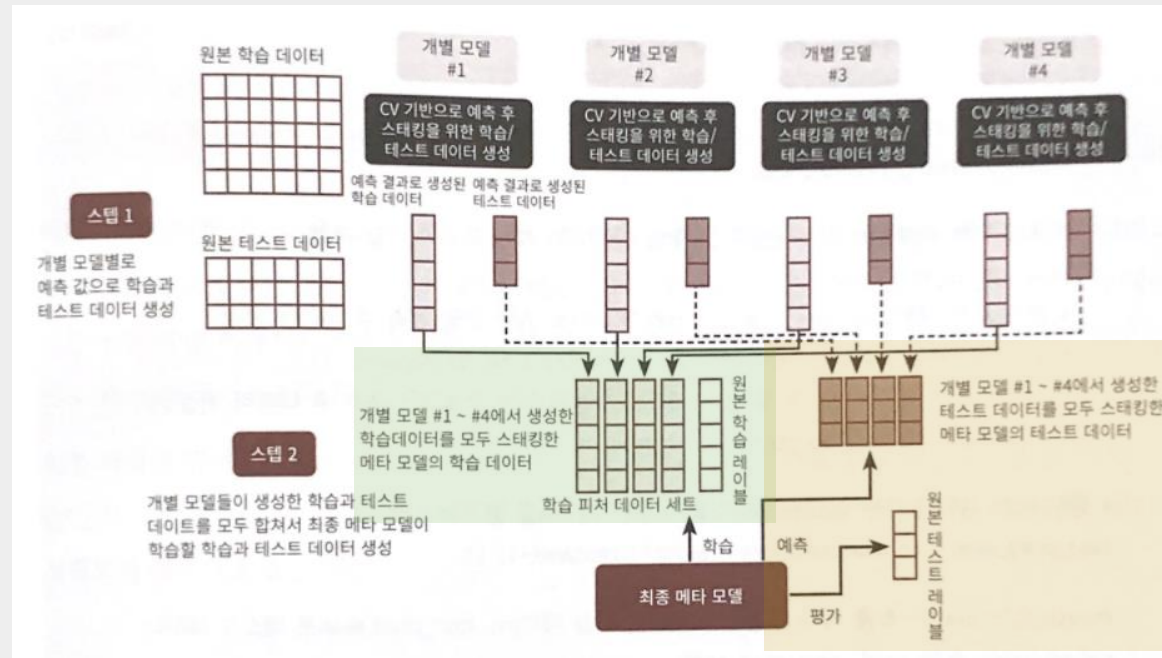
최종 모델의  
학습 데이터

평균값이 최종 모델  
의 테스트 데이터

## 03 스택킹(Stacking) 앙상블

### ○ CV 세트 기반 스택킹 모델

#### STEP 2 (모델 전체 도식화)



6. 최종 학습 데이터 + 원본 Train label 데이터 => 메타모델을 학습

7. 최종 테스트 데이터로 **예측** 수행한 후, 예측 결과를 원본 Test label 데이터와 비교하여 **평가** 진행

# 04 스택킹 앙상블 실습

— Cross Validation (K-Fold) 기반 스택킹 모델 실습

## 04 Cross Validation (K-Fold) 기반 스택킹 모델 실습

# 사이킷런에 내장된 위스콘신 암 데이터 세트를 이용하여 실습을 진행한다.

```
import numpy as np

from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 데이터 로드
cancer_data = load_breast_cancer()

X_data = cancer_data.data
y_label = cancer_data.target

# 학습/테스트 데이터를 80:20 분할
X_train, X_test, y_train, y_test = train_test_split(X_data, y_label, test_size=0.2, random_state=0)
```

# 개별 모델은 KNN, 랜덤 포레스트, 결정 트리, 에이다 부스트 총 4개 모델을 이용하고, 메타 모델은 로지스틱 회귀를 이용한다.

```
# 개별 ML 모델 생성
knn_clf = KNeighborsClassifier(n_neighbors=4)
rf_clf = RandomForestClassifier(n_estimators=100, random_state=0)
dt_clf = DecisionTreeClassifier()
ada_clf = AdaBoostClassifier(n_estimators=100)

# 스택킹으로 만들어진 데이터 세트를 학습 & 예측할 최종 모델 : 로지스틱 회귀
lr_final = LogisticRegression(C=10)
```

해당 실습에서는 파라미터 튜닝을 최적으로 하는 과정은 생략했습니다.

## 04 Cross Validation (K-Fold) 기반 스택킹 모델 실습

# 메타 모델을 위한 학습용, 테스트용 데이터 세트를 생성해주는 함수를 작성해보자! 개별 모델들이 호출할 함수이다 ☺

```
### step 1
from sklearn.model_selection import KFold
from sklearn.metrics import mean_absolute_error

# 개별 기반 모델에서 최종 메타모델이 사용할 학습 및 테스트용 데이터를 생성하기 위한 함수.
def get_stacking_base_datasets(model, X_train_n, y_train_n, X_test_n, n_folds):
    # 지정된 n_folds값으로 KFold 분할
    kf = KFold(n_splits=n_folds, random_state=0)
    # 추후에 메타 모델이 사용할 '학습/테스트 데이터' 반환을 위한 넘파이 배열 초기화
    train_fold_pred = np.zeros((X_train_n.shape[0], n_folds)) # 개별 모델들의 검증 데이터에 대한 예측값 (메타모델의 학습 데이터)
    test_pred = np.zeros((X_test_n.shape[0], n_folds)) # 개별 모델들의 원본 테스트 데이터에 대한 예측값 (메타모델의 테스트 데이터)
    print(model.__class__.__name__, ' model 시작 ')

    for folder_counter, (train_index, valid_index) in enumerate(kf.split(X_train_n)): # Kfold 갯수만큼 반복
        # 인덱스를 이용하여 원본 학습 데이터를 학습/검증 폴드 데이터로 2:1 분할
        print('###폴드 세트: ', folder_counter, ' 시작 ')
        X_tr = X_train_n[train_index] # 학습 피쳐 데이터
        y_tr = y_train_n[train_index] # 학습 레이블 데이터
        X_te = X_train_n[valid_index] # 검증 피쳐 데이터

        # 폴드 세트 내부에서 분할된 학습 데이터로 기반 모델이 학습을 수행한다.
        model.fit(X_tr, y_tr)

        # 폴드 세트 내부에서 분할된 검증 데이터에 대해 기반 모델이 예측한 후, 예측값을 해당 인덱스에 저장한다.
        train_fold_pred[valid_index, :] = model.predict(X_te).reshape(-1,1)

        # 폴드 세트 내부에서 학습되어진 기반 모델이 원본 테스트 데이터에 대해 예측하고 예측값을 저장한다.
        test_pred[:, folder_counter] = model.predict(X_test_n)

    # 기반 모델이 원본 테스트 데이터를 예측한 데이터셋들의 평균값이 최종 테스트 데이터가 된다.
    test_pred_mean = np.mean(test_pred, axis=1).reshape(-1,1)

    #train_fold_pred는 최종 메타 모델이 사용하는 학습 데이터, test_pred_mean은 테스트 데이터
    return train_fold_pred, test_pred_mean
```

지정한 폴드 수만큼 반복하면서  
지정된 인덱스 내 학습/검증 데이터로 학습 및  
예측을 수행!

\* folder\_counter : 폴드 개수 (0~6)

검증 폴드 데이터에 대한 예측 값

↓  
메타 모델의 학습 데이터

원본 테스트 데이터에 대한 예측 값

↓  
메타 모델의 테스트 데이터

- train\_fold\_pred : 기반 모델이 검증 폴드 데이터에 대해 예측한 값
- test\_pred : 기반 모델이 원본 테스트 데이터에 대해 예측한 값
- test\_pred\_mean : test\_pred의 평균값

## 04 Cross Validation (K-Fold) 기반 스택킹 모델 실습

# 방금 작성한 get\_stacking\_base\_datasets() 함수를 각 기반 모델별로 호출해서 메타 모델을 위한 새로운 데이터를 생성해보자.

```
# CV스태킹 알고리즘을 각 모델에 적용하여 메타 모델이 추후에 사용할 학습/테스트 데이터 만들기
knn_train, knn_test = get_stacking_base_datasets(knn_clf, X_train, y_train, X_test, 7)
rf_train, rf_test = get_stacking_base_datasets(rf_clf, X_train, y_train, X_test, 7)
dt_train, dt_test = get_stacking_base_datasets(dt_clf, X_train, y_train, X_test, 7)
ada_train, ada_test = get_stacking_base_datasets(ada_clf, X_train, y_train, X_test, 7)
```

- 모델 객체
- 원본 학습 피쳐 데이터
- 원본 학습 라벨 데이터
- 원본 테스트 피쳐 데이터
- K-Fold 갯수

KNeighborsCl	RandomForestClass	DecisionTreeClassifier	AdaBoostClassifier	model 시작
폴드	폴드 세트	폴드 세트: 0	폴드 세트: 0	시작
폴드	폴드 세트	폴드 세트: 1	폴드 세트: 1	시작
폴드	폴드 세트	폴드 세트: 2	폴드 세트: 2	시작
폴드	폴드 세트	폴드 세트: 3	폴드 세트: 3	시작
폴드	폴드 세트	폴드 세트: 4	폴드 세트: 4	시작
폴드	폴드 세트	폴드 세트: 5	폴드 세트: 5	시작
폴드	폴드 세트	폴드 세트: 6	폴드 세트: 6	시작

```
### step 2
# 생성된 개별 모델들의 학습/테스트 데이터를 합쳐주기
Stack_final_X_train = np.concatenate((knn_train, rf_train, dt_train, ada_train), axis=1)
Stack_final_X_test = np.concatenate((knn_test, rf_test, dt_test, ada_test), axis=1)

print('원본 학습 피쳐 데이터 Shape: ', X_train.shape, '원본 테스트 피쳐 데이터 Shape: ', X_test.shape)
print('스태킹 학습 피쳐 데이터 Shape: ', Stack_final_X_train.shape,
      '스태킹 테스트 피쳐 데이터 Shape: ', Stack_final_X_test.shape)
```

원본 학습 피쳐 데이터 Shape: (455, 30) 원본 테스트 피쳐 데이터 Shape: (114, 30)  
스태킹 학습 피쳐 데이터 Shape: (455, 4) 스태킹 테스트 피쳐 데이터 Shape: (114, 4)

이런 형태!

[illegible]

## 04 Cross Validation (K-Fold) 기반 스택킹 모델 실습

# 최종 메타 모델을 학습/예측을 진행하고 정확도를 확인하자.

```
# 메타 모델 학습 & 예측
lr_final.fit(Stack_final_X_train, y_train)
stack_final = lr_final.predict(Stack_final_X_test)

print('최종 메타 모델의 예측 정확도: {0:.4f}'.format(accuracy_score(y_test, stack_final)))
```

스태킹된 학습용 피쳐 데이터 세트와  
원본 학습 레이블 데이터로 학습 진행!

최종 메타 모델의 예측 정확도: 0.9737

실습에서는 개별 모델의 알고리즘에 대해 파라미터 튜닝을 하지 않았지만,  
스태킹을 이루는 모델은 **최적으로 파라미터를 튜닝한 상태에서  
스태킹 모델을 만드는 것이 일반적 !**

- **스태킹 모델의 파라미터 튜닝**은 결국 개별 알고리즘 모델들의 파라미터를 최적으로 튜닝하는 것!
- 스택킹 모델은 분류(Classification) 뿐만 아니라 회귀(Regression)에서도 적용 가능함



Q&A

**THANK  
YOU**

Bitamin 4 조

**김회인 | 김지나 | 문윤지 | 유승길**