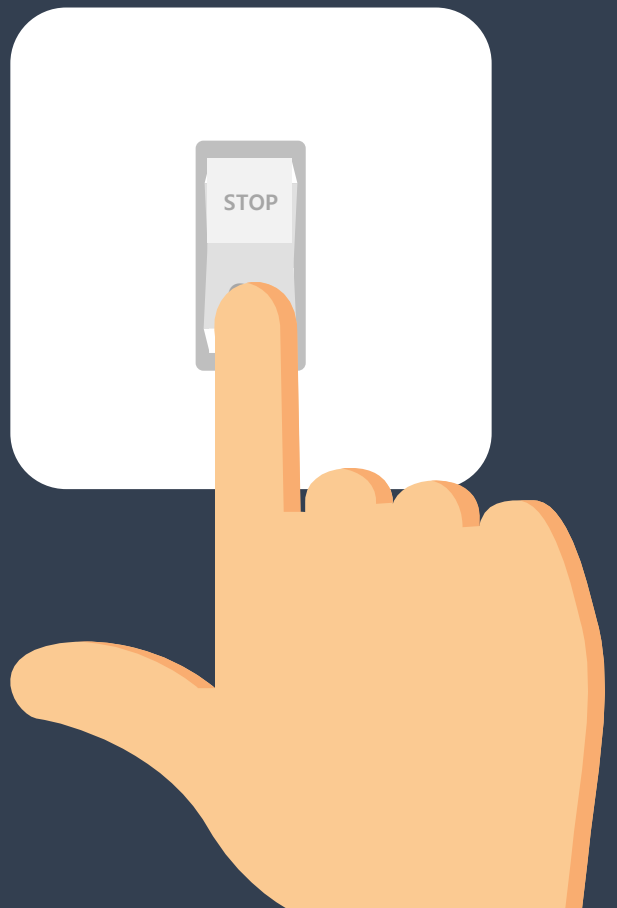
BITAmin

12주차 정규 Session

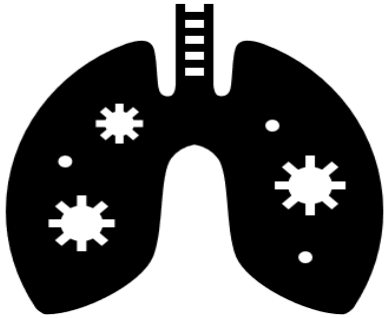


4조

김현우 이수빈 임수진 정성현

목차

1. 불균형 데이터-불균형 데이터의 특징



의료 현장

은행 거래

1. 불균형 데이터-불균형 데이터의 특징

클래스 ={정상, 불량}		예측한 클래스	
		정상	불량
실제 클래스	정상	TN	FP
	불량	FN	TP

$$\text{정밀도}(Precision) = \frac{\text{옳게 분류된 불량 데이터의 수}}{\text{불량으로 예측한 데이터}} = \frac{TP}{FP + TP}$$

$$\text{재현율}(Recall) = \frac{\text{옳게 분류된 불량 데이터의 수}}{\text{실제 불량 데이터의 수}} = \frac{TP}{FN + TP}$$

$$\text{특이도}(Specificity) = \frac{\text{옳게 분류된 정상 데이터의 수}}{\text{실제 정상 데이터의 수}} = \frac{TN}{TN + FP}$$

정밀도가 높다면?→ major를 major로 잘 판단함
정밀도가 낮다면?→major를 minor로 판단함
재현율이 높다면?→ minor를 minor로 잘 판단함
재현율이 낮다면?→minor를 major로 판단함

→ **가장 큰 ISSUE**

1. 불균형 데이터-불균형 데이터의 특징

-MAJOR CLASS 를 잘못 판단했을 때의 영향

<<<MINOR CLASS를 잘못 판단했을 때의 영향

-ACCURACY PARADOX

1. 불균형 데이터-불균형 데이터의 특징

StandardScaler

로그변환

이상치 제거

SMOTE

0. BASIC

1. 불균형 데이터-불균형 데이터의 특징

```
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	0.266151	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	-2.066656	-5.364473	-2.606837	-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

1. 불균형 데이터-불균형 데이터의 특징

```
from sklearn.model_selection import train_test_split

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

데이터 전처리를 위한 함수
- 이후에 옵션을 추가해가며 빠른 전처리를 도움

```
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
```

1. 불균형 데이터-불균형 데이터의 특징

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

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)
```

```
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
```

IR: Imbalanced Ratio: Majority Class/Minority Class
→ IR 이 매우 큼
→ Class-Imbalanced Dataeset

1. 불균형 데이터-불균형 데이터의 특징

```
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]
```

```
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))
```

1. 불균형 데이터-불균형 데이터의 특징

```
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)
```

Default 값이 바뀜에 따라 꼭 지정해줘야 함!

```
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)
```

1. 불균형 데이터-불균형 데이터의 특징

```
get_clf_eval(y_test, lr_pred, lr_pred_proba)
```

```
오차 행렬
[[85283    12]
 [   56    92]]
정확도: 0.9992, 정밀도: 0.8846, 재현율: 0.6216, F1: 0.7302, AUC: 0.9599
```

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

1. 불균형 데이터-불균형 데이터의 특징

데이터 가공	머신러닝 알고리즘	평가지표		
		정밀도	재현율	ROC-AUC
원본 데이터	로지스틱 회귀	0.8846	0.6216	0.9599
	LightGBM	0.9573	0.7568	0.9790

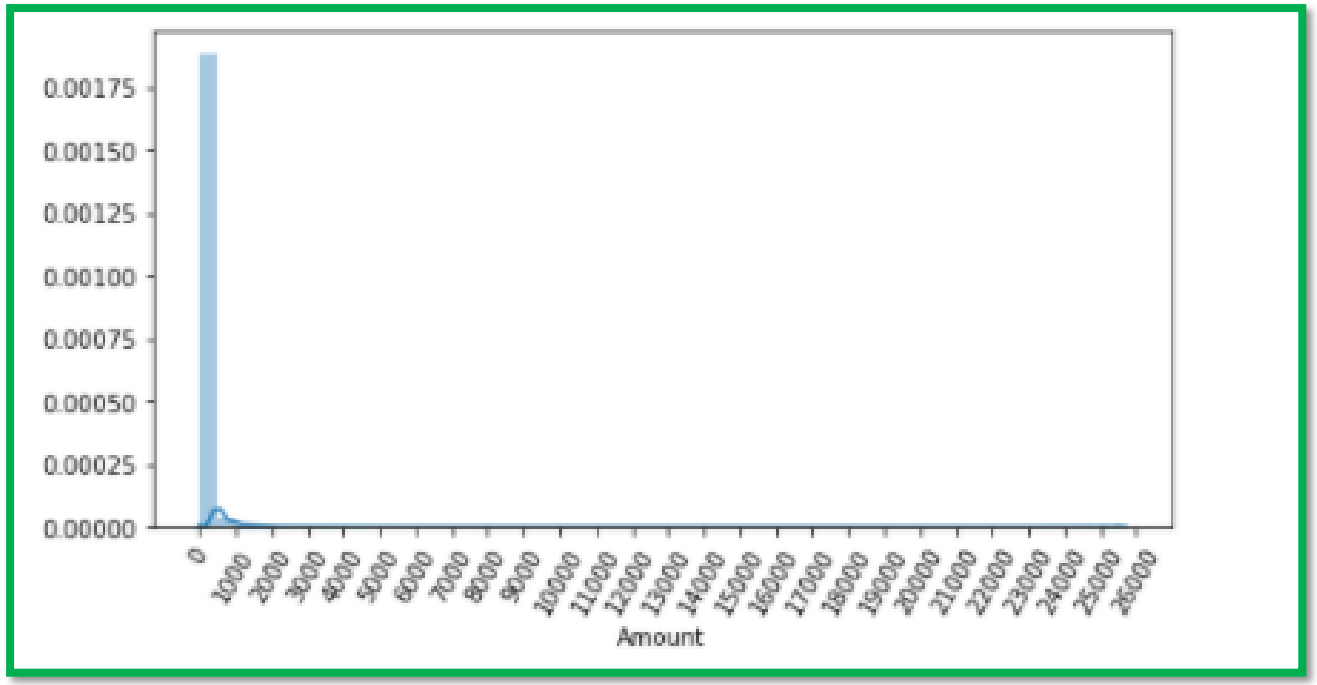
1. StandardScaler

1. 불균형 데이터-불균형 데이터의 특징

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

Amount: 정상/사기 트랜잭션을 결정하는 매우 중요한 속성

StandardScaler를 통해 이 피처를 정규분포 형태로 변환



1. 불균형 데이터-불균형 데이터의 특징

```
from sklearn.preprocessing import StandardScaler

def get_preprocessed_df(df=None):
    df_copy=df.copy()
    scaler=StandardScaler()
    amount_n=scaler.fit_transform(df_copy['Amount'].values.reshape(-1, 1))
    df_copy.insert(0, 'Amount_scaled', amount_n)
    df_copy.drop(['Amount', 'Time'], axis=1, inplace=True)
    return df_copy
```

```
X_train, X_test, y_train, y_test=get_train_test_dataset(card_df)
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('### LGBM 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)
```

1. 불균형 데이터-불균형 데이터의 특징

```
오차 행렬
[[85281    14]
 [   58    90]]
정확도: 0.9992, 정밀도: 0.8654, 재현율: 0.6081, F1: 0.7143, AUC: 0.9702
### LGBM 예측 성능 ###
오차 행렬
[[85290     5]
 [   37   111]]
정확도: 0.9995, 정밀도: 0.9569, 재현율: 0.7500, F1: 0.8409, AUC: 0.9779
```

1. 불균형 데이터-불균형 데이터의 특징

데이터 가공	머신러닝 알고리즘	평가지표		
		정밀도	재현율	ROC-AUC
원본 데이터	로지스틱 회귀	0.8846	0.6216	0.9599
	LightGBM	0.9573	0.7568	0.9790
StandardScaler	로지스틱 회귀	**0.8654**	**0.6081**	**0.9702**
	LightGBM	**0.9569**	**0.7500**	**0.9779**

2. 로그 변환

데이터 분포가 심하게 왜곡되어 있을 경우 적용하는 기법

원래 값을 log값으로 변환하기에 상대적으로 적은 값으로 모델을 판단할 수 있음

하양이 5장에서 더 다룰 예정

1. 불균형 데이터-불균형 데이터의 특징

```
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(['Amount', 'Time'], axis=1, inplace=True)
    return df_copy
```

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

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

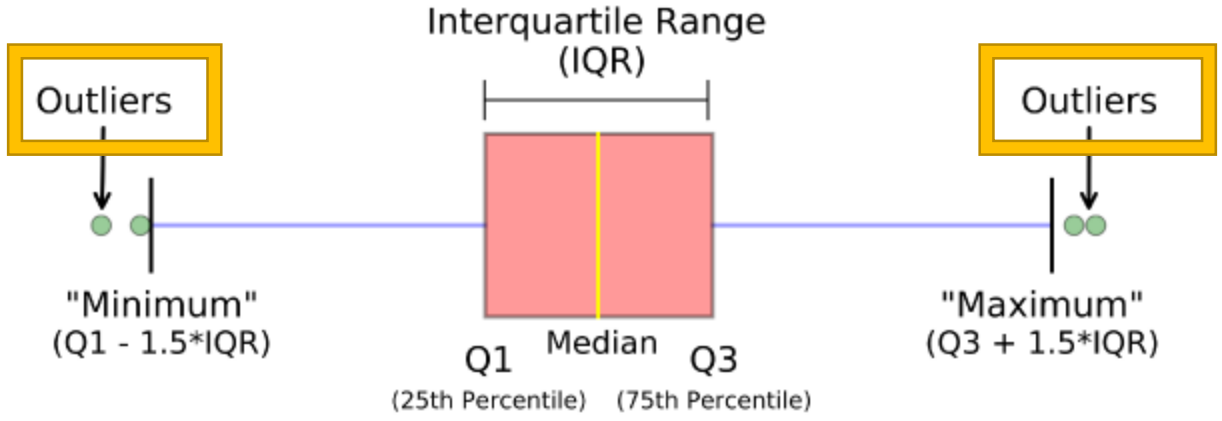
1. 불균형 데이터-불균형 데이터의 특징

```
### 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
```

1. 불균형 데이터-불균형 데이터의 특징

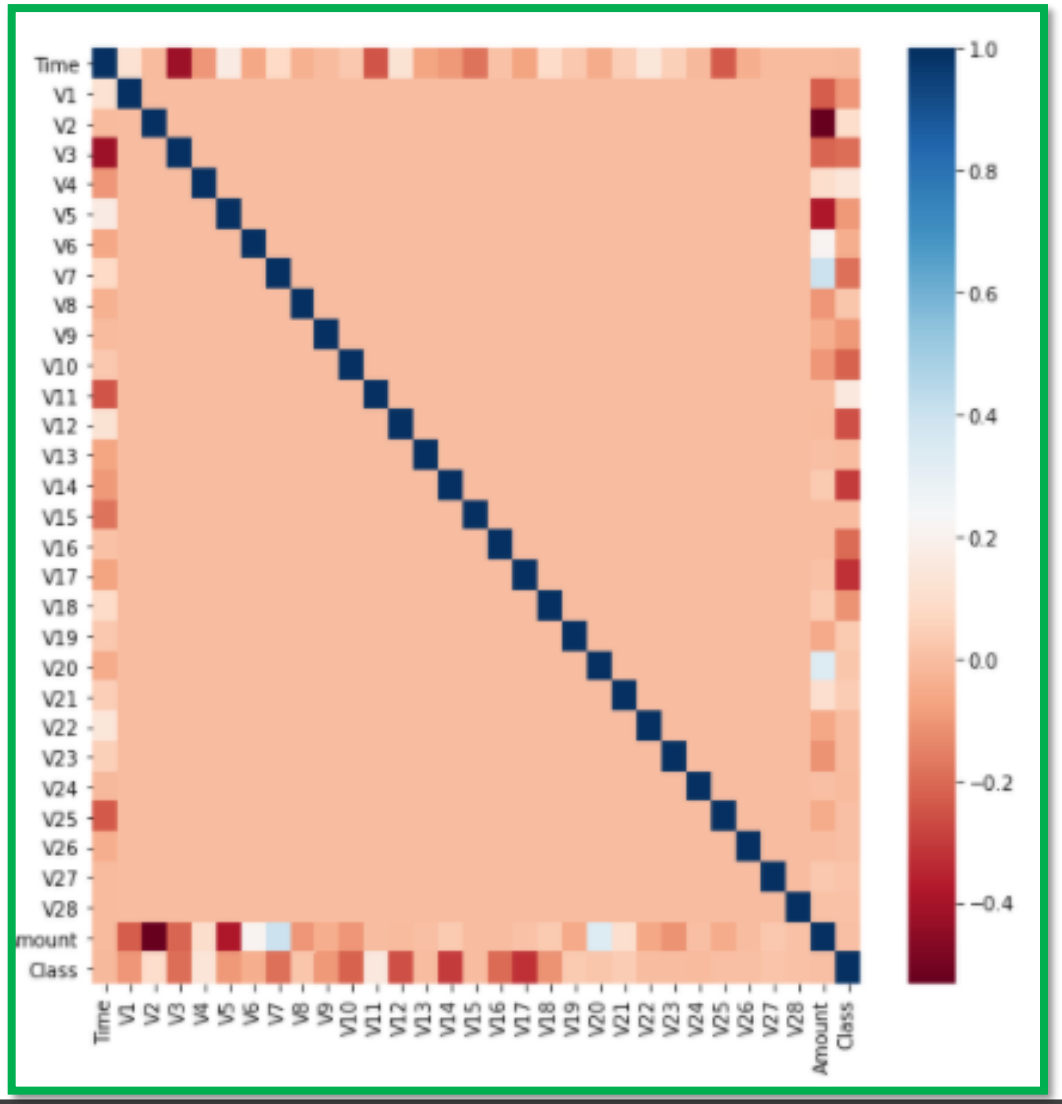
데이터 가공	머신러닝 알고리즘	평가지표		
		정밀도	재현율	ROC-AUC
원본 데이터	로지스틱 회귀	0.8846	0.6216	0.9599
	LightGBM	0.9573	0.7568	0.9790
StandardScaler	로지스틱 회귀	0.8654	0.6081	0.9702
	LightGBM	0.9569	0.7500	0.9779
로그 변환	**로지스틱 회귀**	**0.8812**	**0.6014**	**0.9727**
	LightGBM	**0.9576**	**0.7635**	**0.9796**

3. 이상치 제거


Interquartile Range
(IQR)
Outliers
Outliers
"Minimum"
(Q1 - 1.5*IQR)
Q1
(25th Percentile)
Median
Q3
(75th Percentile)
"Maximum"
(Q3 + 1.5*IQR)

1. 불균형 데이터-불균형 데이터의 특징

```
plt.figure(figsize=(9, 9))
corr=card_df.corr()
sns.heatmap(corr, cmap='RdBu')
```



1. 불균형 데이터-불균형 데이터의 특징

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

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

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

1. 불균형 데이터-불균형 데이터의 특징

```
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(['Amount', 'Time'], axis=1, inplace=True)
    outlier_index=get_outlier(df=df_copy, columns='V14', weight=1.5)
    df_copy.drop(outlier_index, axis=0, inplace=True)
    return df_copy
```

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

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

1. 불균형 데이터-불균형 데이터의 특징

```
### 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
```

1. 불균형 데이터-불균형 데이터의 특징

데이터 가공	머신러닝 알고리즘	평가지표		
		정밀도	재현율	ROC-AUC
원본 데이터	로지스틱 회귀	0.8846	0.6216	0.9599
	LightGBM	0.9573	0.7568	0.9790
StandardScaler	로지스틱 회귀	0.8654	0.6081	0.9702
	LightGBM	0.9569	0.7500	0.9779
로그 변환	로지스틱 회귀	0.8812	0.6014	0.9727
	LightGBM	0.9576	0.7635	0.9796
이상치 제거	**로지스틱 회귀**	**0.8750**	**0.6712**	**0.9743**
	LightGBM	**0.9603**	**0.8288**	**0.9780**

4. SMOTE

- pip: `pip install -U imbalanced-learn`
- anaconda: `conda install -c glemaitre imbalanced-learn`

1. 불균형 데이터-불균형 데이터의 특징

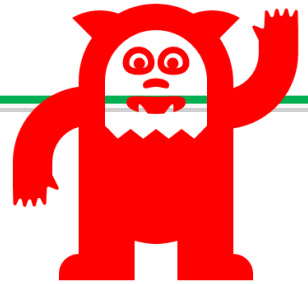
```
from imblearn.over_sampling import SMOTE
```

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

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

```
lr_clf=LogisticRegression()
get_model_train_eval(lr_clf, ftr_train=X_train_over, ftr_test=X_test,
                     tgt_train=y_train_over, tgt_test=y_test)
```

1. 불균형 데이터-불균형 데이터의 특징



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

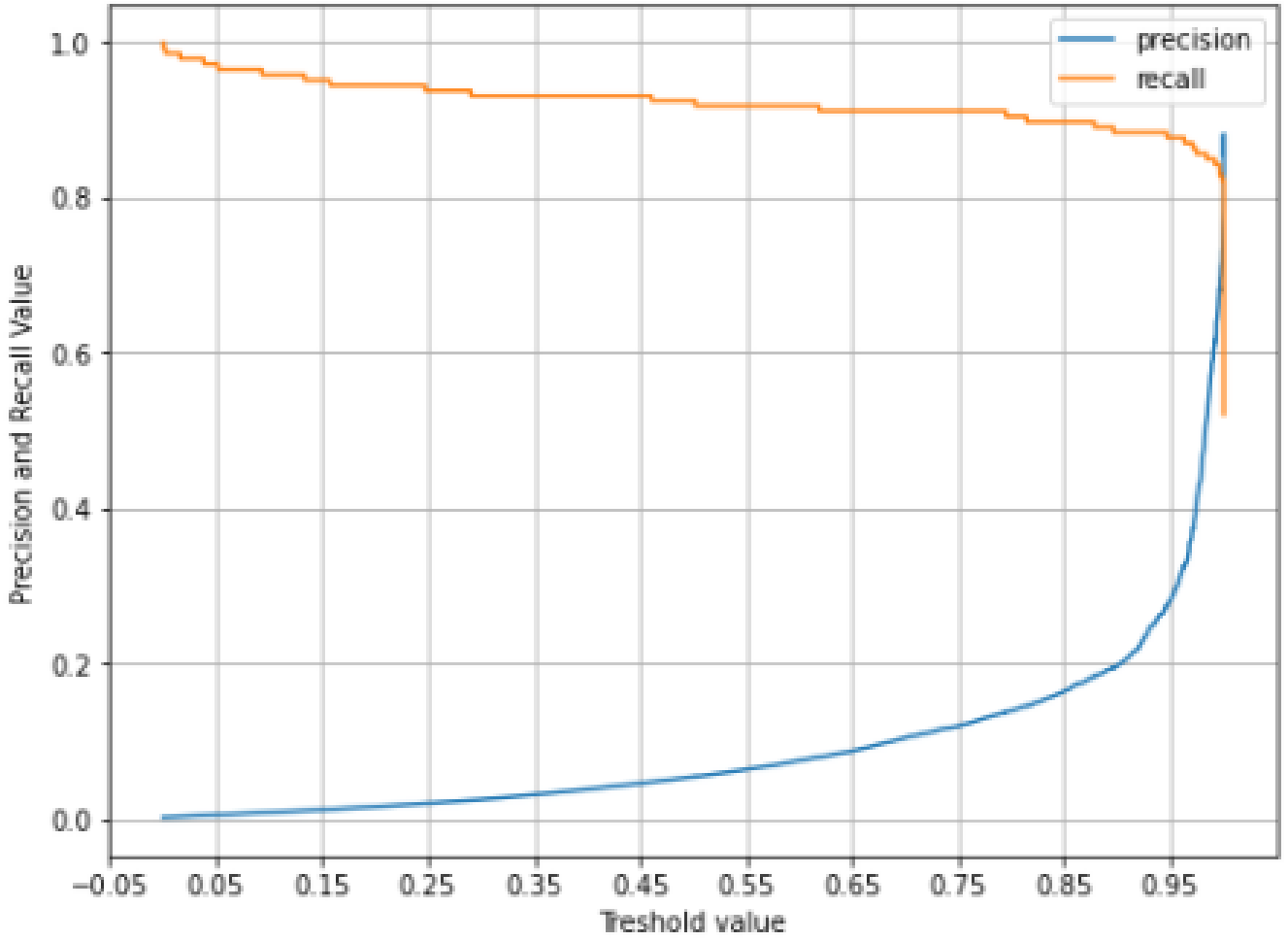
로지스틱 회귀 모델이 오버 샘플링으로 인해 실제 원본 데이터의 유형보다 너무나 많은 Class=1 데이터를 학습하면서 실제 데이터 세트에서 Class=1 예측을 지나치게 적용해 정밀도가 급격히 떨어지게 된 것

1. 불균형 데이터-불균형 데이터의 특징

```
from sklearn.preprocessing import Binarizer

def precision_recall_curve_plot(y_test, pred_proba):
    precisions, recalls, thresholds=precision_recall_curve(y_test, pred_proba)
    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('Treshold value'); plt.ylabel('Precision and Recall Value')
    plt.legend();plt.grid()
    plt.show()

precision_recall_curve_plot(y_test, lr_clf.predict_proba(X_test)[:,1])
```

분류 결정 임계값에 따른 정밀도와 재현율 곡선을 통해 SMOTE로 학습된 로지스틱 회귀 모델에 어떠한 문제가 발생했는지 확인

1. 불균형 데이터-불균형 데이터의 특징



〈임계값 0.99를 기준〉

이상: 재현율 높고, 정밀도 낮음

이하: 재현율 낮아지고 정밀도 높아짐

사용 X

1. 불균형 데이터-불균형 데이터의 특징

```
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)
```

```
오차 행렬
[[85283    12]
 [   22   124]]
정확도: 0.9996, 정밀도: 0.9118, 재현율: 0.8493, F1: 0.8794, AUC: 0.9814
```

1. 불균형 데이터-불균형 데이터의 특징

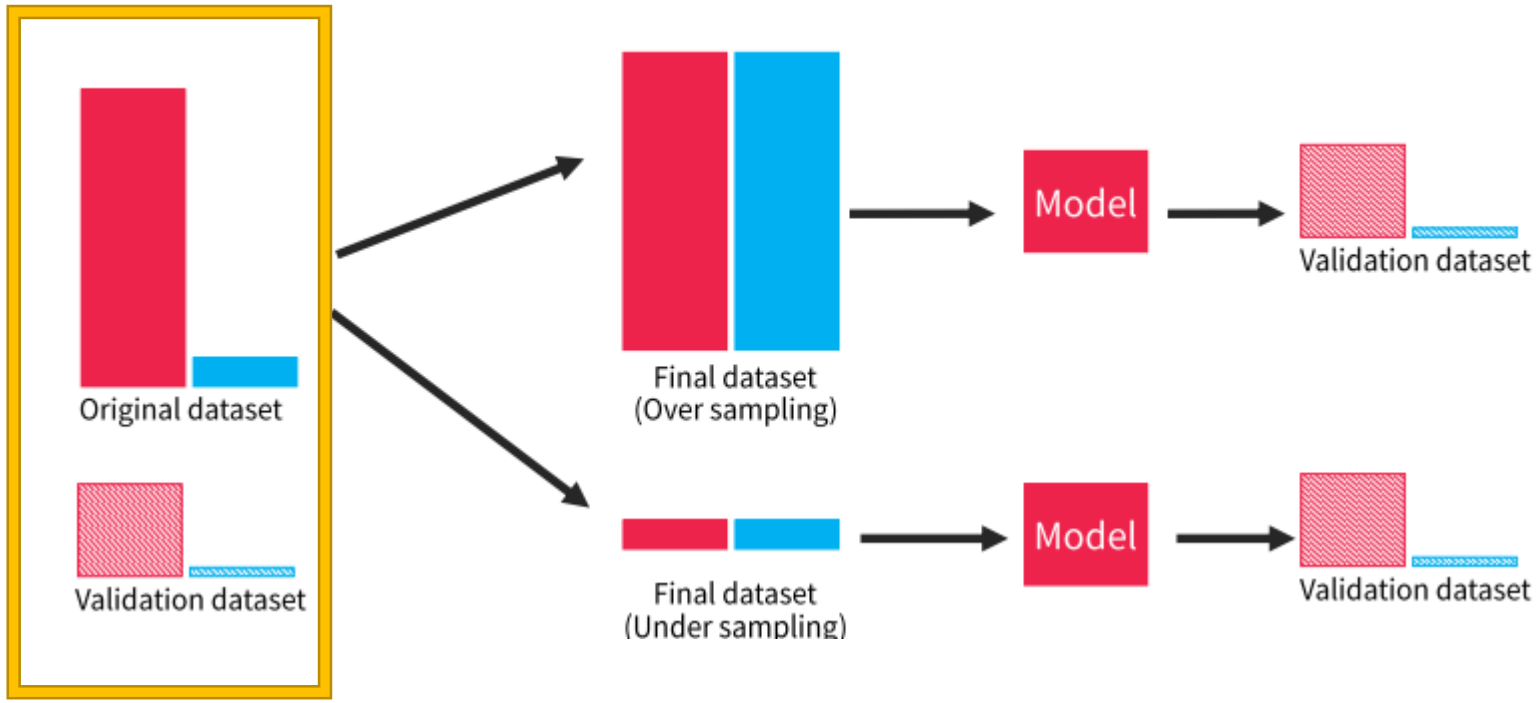
데이터 가공	머신러닝 알고리즘	평가지표		
		정밀도	재현율	ROC-AUC
원본 데이터	로지스틱 회귀	0.8846	0.6216	0.9599
	LightGBM	0.9573	0.7568	0.9790
StandardScaler	로지스틱 회귀	0.8654	0.6081	0.9702
	LightGBM	0.9569	0.7500	0.9779
로그 변환	로지스틱 회귀	0.8812	0.6014	0.9727
	LightGBM	0.9576	0.7635	0.9796
이상치 제거	로지스틱 회귀	0.8750	0.6712	0.9743
	LightGBM	0.9603	0.8288	0.9780
SMOTE	**로지스틱 회귀**	**0.0542**	**0.9247**	**0.9737**
	LightGBM	**0.9118**	**0.8493**	**0.9814**

1. 불균형 데이터-불균형 데이터의 특징

Random Sampling

Cost-Sensitive Learning

1. 불균형 데이터-불균형 데이터의 특징



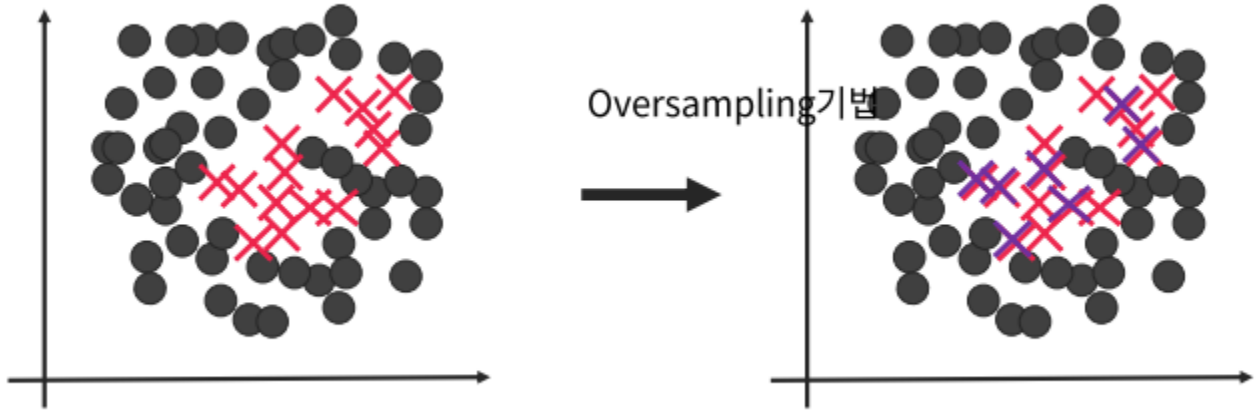
반드시

학습 데이터에 대해서만 over/under 샘플링을 진행해야 함.

검증 데이터에 대해서 over/under 샘플링을 할 경우 올바른 검증을 할 수 없음.

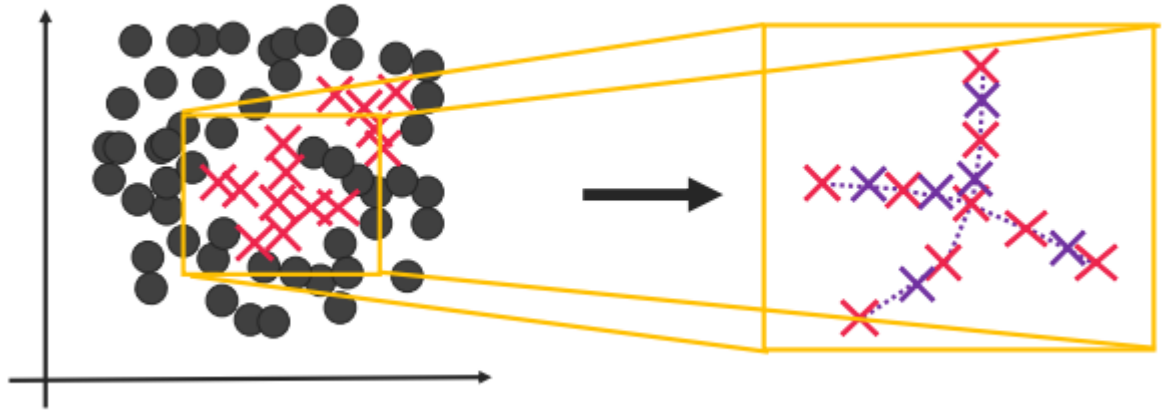
1. 불균형 데이터-불균형 데이터의 특징

- Random Over Sampling



1. 불균형 데이터-불균형 데이터의 특징

- **SMOTE(Synthetic Minority Over-sampling Technique)**

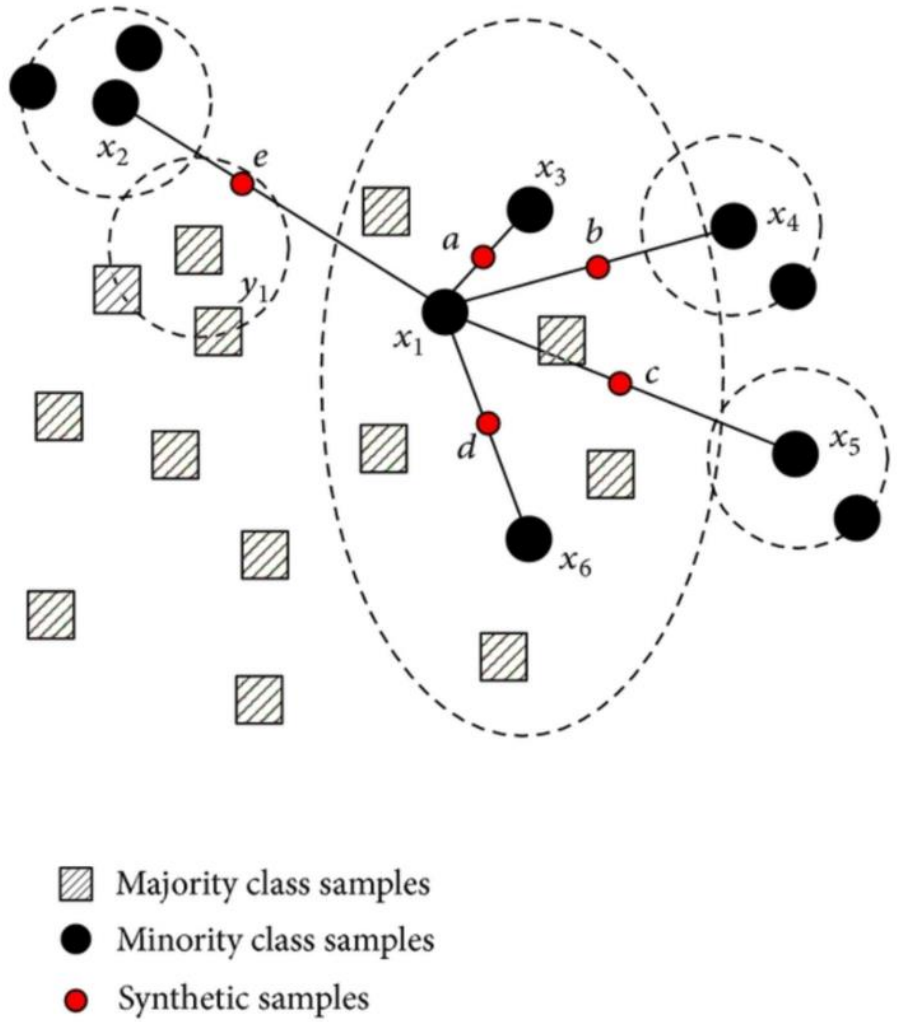


Algorithm *SMOTE*(T, N, k)
Input: Number of minority class samples T; Amount of SMOTE N%; Number of nearest neighbors k
Output: $(N/100) * T$ synthetic minority class samples

(∗ *If N is less than 100%, randomize the minority class samples as only a random percent of them will be SMOTEd.* ∗)
if $N < 100$
  then Randomize the T minority class samples
    $T = (N/100) * T$
    $N = 100$
endif
$N = (int)(N/100)$ (∗ *The amount of SMOTE is assumed to be in integral multiples of 100.* ∗)
k = Number of nearest neighbors
numattrs = Number of attributes
Sample[ ][ ]: array for original minority class samples
newindex: keeps a count of number of synthetic samples generated, initialized to 0
Synthetic[ ][ ]: array for synthetic samples
(∗ *Compute k nearest neighbors for each minority class sample only.* ∗)
for $i \leftarrow 1$ **to** T
    Compute k nearest neighbors for i, and save the indices in the *nnarray*
    Populate(N, i, *nnarray*)
endfor

Populate(N, i, *nnarray*) (∗ *Function to generate the synthetic samples.* ∗)
while $N \neq 0$
    Choose a random number between 1 and k, call it nn. This step chooses one of the k nearest neighbors of i.
    for $attr \leftarrow 1$ **to** *numattrs*
      Compute: $dif = Sample[nnarray[nn]][attr] - Sample[i][attr]$
      Compute: gap = random number between 0 and 1
      $Synthetic[newindex][attr] = Sample[i][attr] + gap * dif$
    endfor
    $newindex{+}{+}$
    $N = N - 1$
endwhile
return (∗ *End of Populate.* ∗)
End of Pseudo-Code.

1. 불균형 데이터-불균형 데이터의 특징

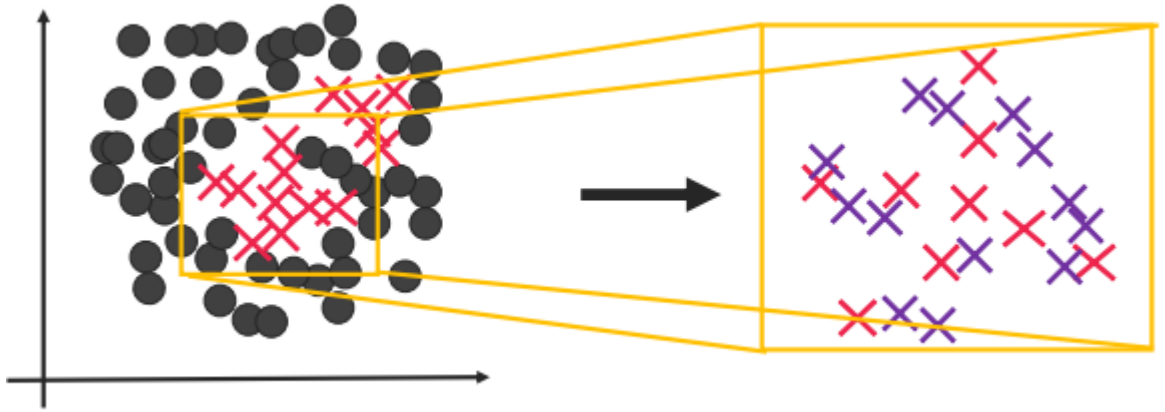


1. 소수 클래스에서 각각의 샘플들의 KNN을 찾는다.

2. 각각의 군집을 선으로 이은 후, 선 위의 임의의 점에 새로운 점을 생성한다.

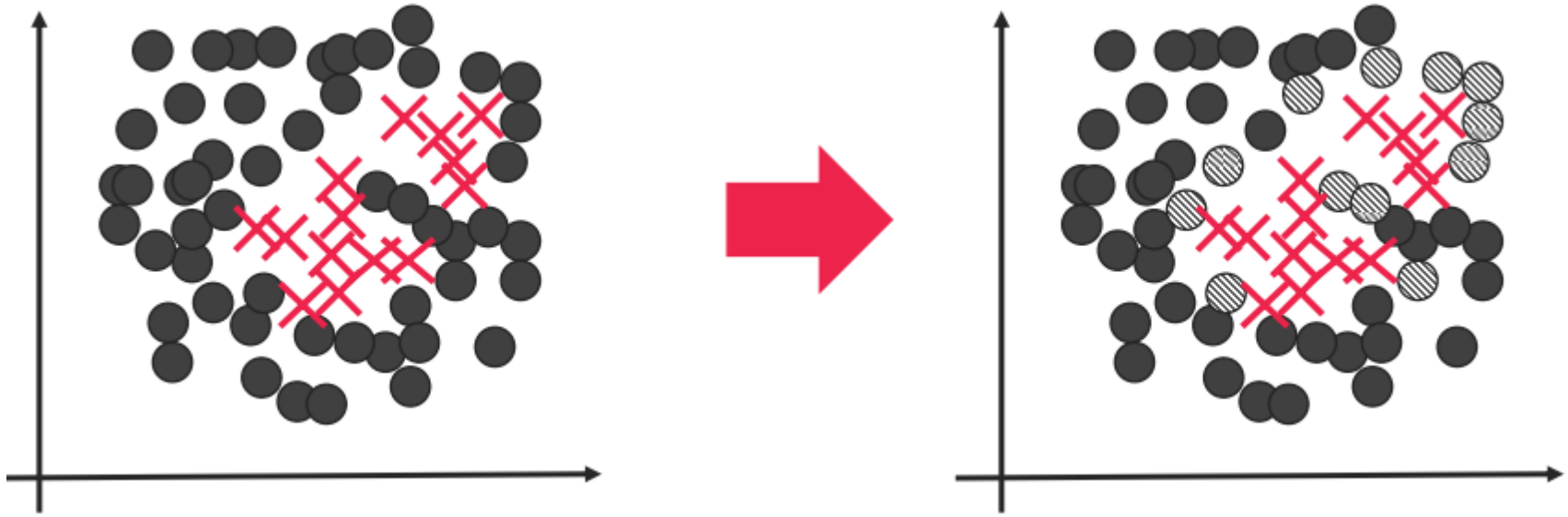
1. 불균형 데이터-불균형 데이터의 특징

- BLSMOTE(Border Line SMOTE)



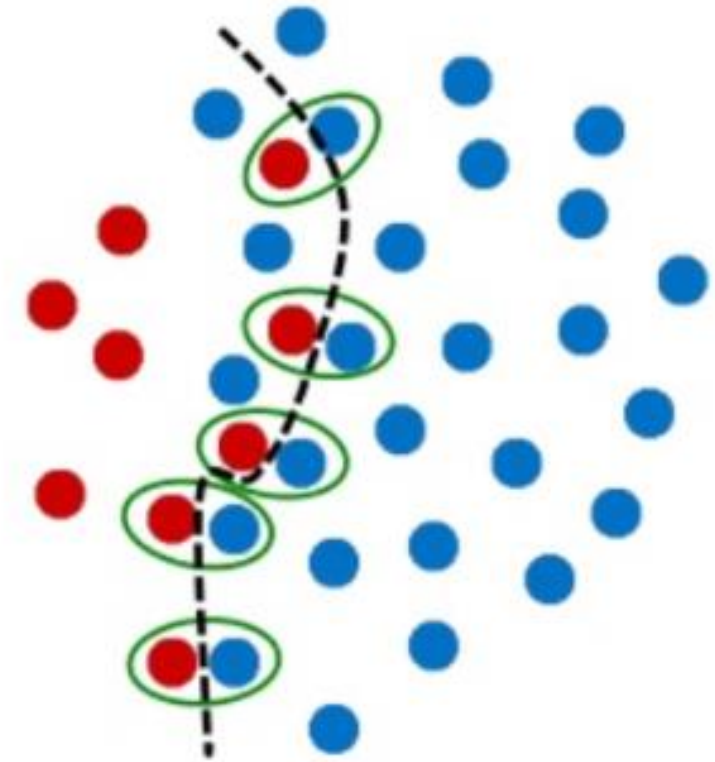
1. 불균형 데이터-불균형 데이터의 특징

- Tomek Links



- 이 외에도 Easy Ensemble, Balance Cascade와 같은 기법이 있음

1. 불균형 데이터-불균형 데이터의 특징



1. 서로 다른 클래스의 데이터를 모두 잇는다.

2. 임의의 점 k 에 그은 선분의 길이보다 짧은 거리에 있는 점과의 연결을 토맥링크라고 한다.

3. 토맥링크 중 Major class를 삭제한다.

1. 불균형 데이터-불균형 데이터의 특징

Over Sampling

→데이터의 크기를 늘리기에 시간이 오래 걸림

→over-generating 현상

→과적합의 이슈가 있음

Over Sampling을 통해 의사 결정 경계가 과하게 커지며, 기존의 데이터 분포가 과하게 왜곡되는 경우 발생

이러한 이슈를 해결하기 위해 현재에는 DBSM과 같은 hybrid resampling 기법이 발달

Under Sampling

→ 학습에 필요하지 않은 데이터의 수를 줄임으로써 학습속도 향상

→ Decision Boundary 부근에 있는 데이터를 제거할 경우 학습에 악영향

2. 스태킹 앙상블

스태킹

* 배깅 및 부스팅과의 공통점

  - 개별적인 여러 알고리즘을 서로 결합해 예측 결과를 도출

* 배깅 및 부스팅과의 차이점

  - 알고리즘으로 예측한 데이터를 기반으로 다시 예측을 수행

    ! 즉, 개별 알고리즘의 예측 결과 데이터세트를 최종적인 메타 데이터세트로 만들어
    별도의 ML 알고리즘으로 최종 학습을 수행하고 테스트 데이터를 기반으로 다시 최종 예측을 수행하는 방식

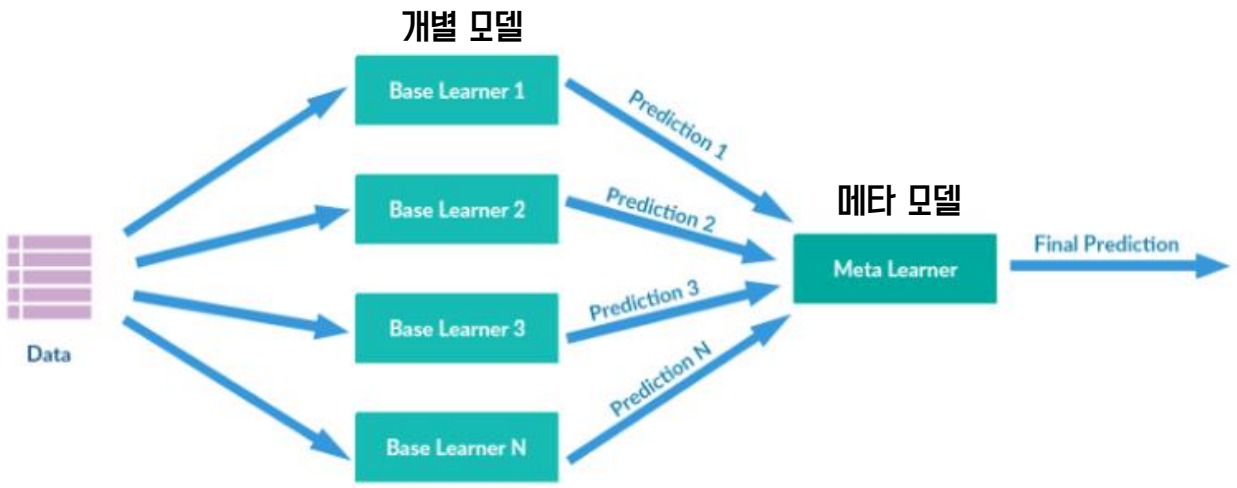
-> 메타 모델 : 개별 모델의 예측된 데이터 세트를 다시 기반으로 하여 학습하고 예측하는 방식

2. 스태킹 앙상블

* 두 종류의 모델 필요

1) 개별적인 기반 모델

2) 이 개별 기반 모델의 예측 데이터를 학습 데이터로 만들어서 학습하는 최종 메타 모델

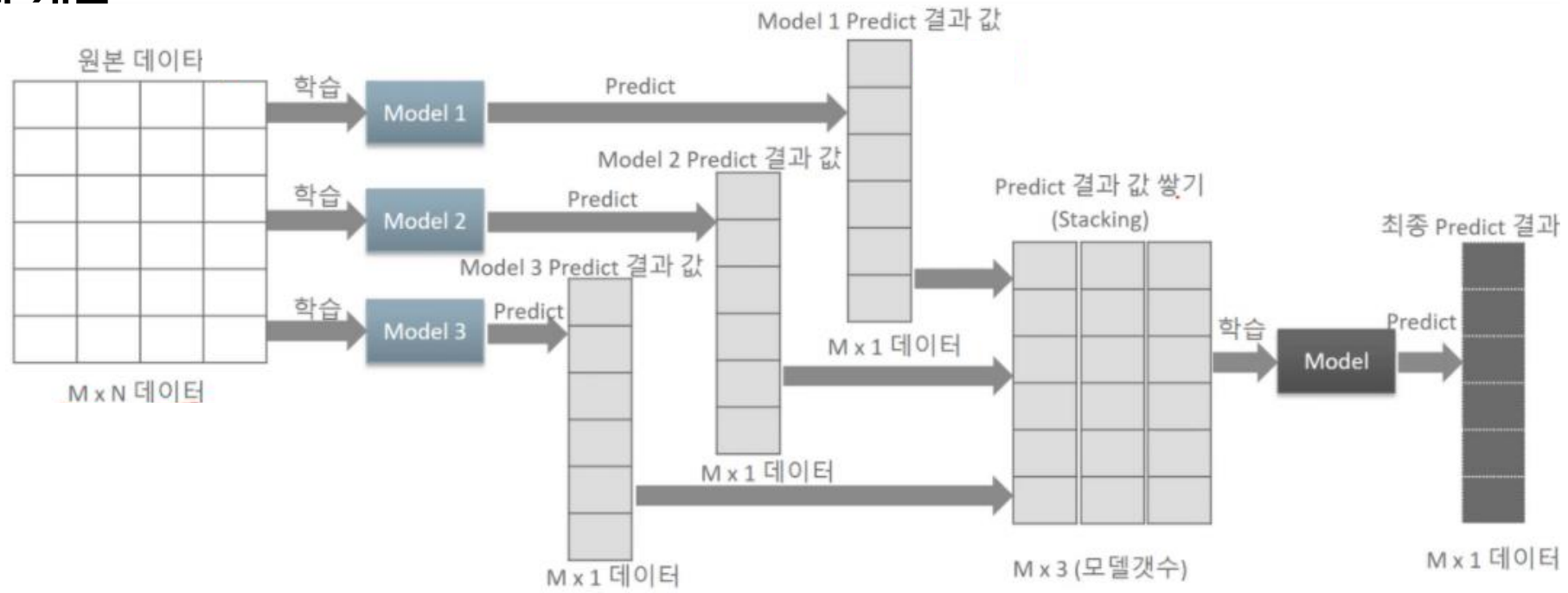


핵심 ! 여러 개별 모델의 예측 데이터를 각각 스태킹 형태로 결합해

최종 메타 모델의 학습용 피처 데이터 세트와 테스트용 피처 데이터 세트를 만드는 것

2. 스태킹 앙상블

* 스태킹의 개념



M개의 로우, N개의 피처(칼럼)을 가진 데이터세트에 스태킹 앙상블을 적용

3가지 모델로 각 모델을 학습한 뒤 예측을 수행하면 M개의 로우를 가진 1개의 레이블 값 도출

-> 모델별로 도출된 예측 레이블 값을 다시 합해서(스태킹)

-> 스태킹된 새로운 데이터세트에 대해 최종 모델 적용하여 최종 예측

2. 스태킹 앙상블

* 예제 (위스콘신 암 데이터 세트)

```
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

cancer_data = load_breast_cancer()

X_data = cancer_data.data
y_label = cancer_data.target

X_train , X_test , y_train , y_test = train_test_split(X_data , y_label ,
                                                       test_size=0.2 , random_state=0)
```

스태킹에 사용될 머신러닝 알고리즘 클래스 생성

```
# 개별 ML 모델을 위한 Classifier 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)

# 최종 Stacking 모델을 위한 Classifier생성.
lr_final = LogisticRegression(C=10)
```

```
# 개별 모델들을 학습.
knn_clf.fit(X_train, y_train)
rf_clf.fit(X_train , y_train)
dt_clf.fit(X_train , y_train)
ada_clf.fit(X_train, y_train)
```

```
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=100, random_state=None)
```

-> 개별 모델 : KNN, 랜덤 포레스트, 결정 트리, 에이다부스트

최종 모델 : 로지스틱 회귀

2. 스태킹 앙상블

```
# 학습된 개별 모델들이 각자 반환하는 예측 데이터 셋을 생성하고 개별 모델의 정확도 측정.
knn_pred = knn_clf.predict(X_test)
rf_pred = rf_clf.predict(X_test)
dt_pred = dt_clf.predict(X_test)
ada_pred = ada_clf.predict(X_test)

print('KNN 정확도: {0:.4f}'.format(accuracy_score(y_test, knn_pred)))
print('랜덤 포레스트 정확도: {0:.4f}'.format(accuracy_score(y_test, rf_pred)))
print('결정 트리 정확도: {0:.4f}'.format(accuracy_score(y_test, dt_pred)))
print('에이다부스트 정확도: {0:.4f} :'.format(accuracy_score(y_test, ada_pred)))
```

```
KNN 정확도: 0.9211
랜덤 포레스트 정확도: 0.9649
결정 트리 정확도: 0.9035
에이다부스트 정확도: 0.9561 :
```

-> 개별 모델의 예측 정확도

```
pred = np.array([knn_pred, rf_pred, dt_pred, ada_pred])
print(pred.shape)

# transpose를 이용해 행과 열의 위치 교환. 컬럼 레벨로 각 알고리즘의 예측 결과를 피처로 만듦.
pred = np.transpose(pred)
print(pred.shape)
```

```
(4, 114)
(114, 4)
```

```
lr_final.fit(pred, y_test)
final = lr_final.predict(pred)

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

```
최종 메타 모델의 예측 정확도: 0.9737
```

스태킹 적용한 결과, 개별 모델 정확도보다 향상됨

! but 최종 학습할 때 테스트용 레이블 데이터 세트를 기반으로 학습함으로 인해 과적합 문제 발생 가능

해결 -> CV 세트 기반 스태킹 모델

-> 예측 데이터로 생성된 데이터 세트를 기반으로 최종 메타 모델인 로지스틱 회귀 학습, 예측 정확도 측정

2. 스태킹 앙상블

* CV 세트 기반의 스태킹

: 과적합 개선을 위해 최종 메타 모델을 위한 데이터 세트를 만들 때 교차 검증 기반으로 예측된 결과 데이터 세트를 이용

* 2단계의 스텝

STEP 1

1) 각 모델별로 원본 학습/테스트 데이터를 예측한 결과 값을 기반으로
   메타 모델을 위한 학습용/테스트용 데이터를 생성

STEP 2

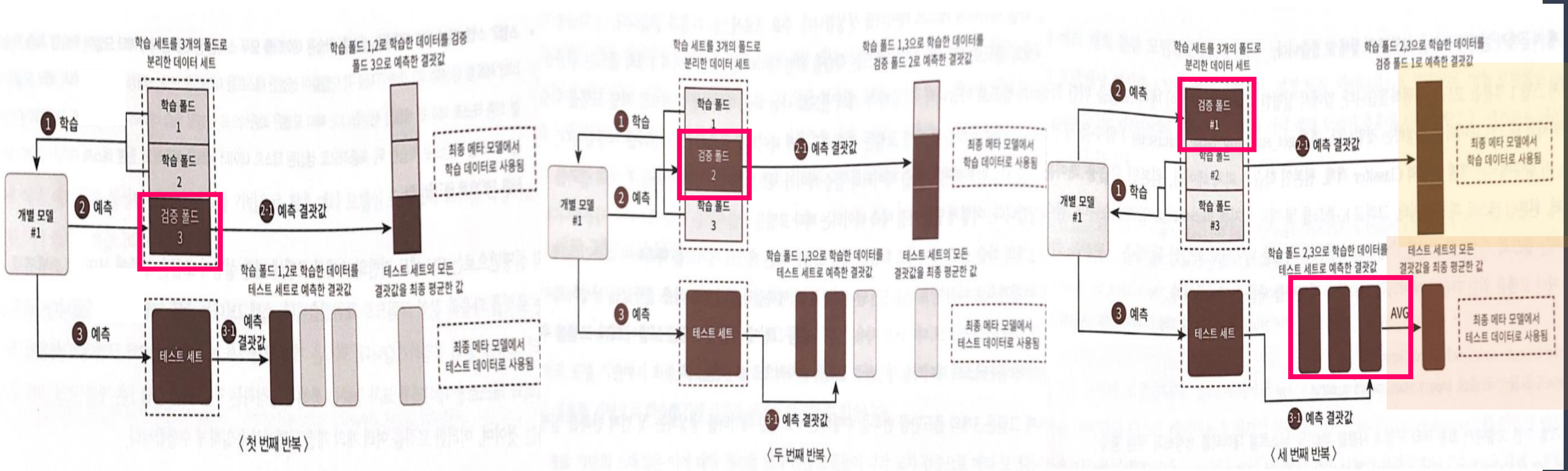
2) STEP 1에서 개별 모델들이 생성한 학습용 데이터를 모두 스태킹 형태로 합쳐서
   메타 모델이 학습할 최종 학습용 데이터 세트 생성

3) 마찬가지로 각 모델들이 생성한 테스트용 데이터를 모두 스태킹 형태로 합쳐서
   메타 모델이 예측할 최종 테스트 데이터 세트를 생성

4) 최종적으로 생성된 학습 데이터 세트와 원본 학습 데이터의 레이블 데이터를 기반으로 학습
   -> 최종적으로 생성된 테스트 데이터 세트 예측, 원본 테스트 데이터의 레이블 데이터를 기반으로 평가

2. 스태킹 앙상블

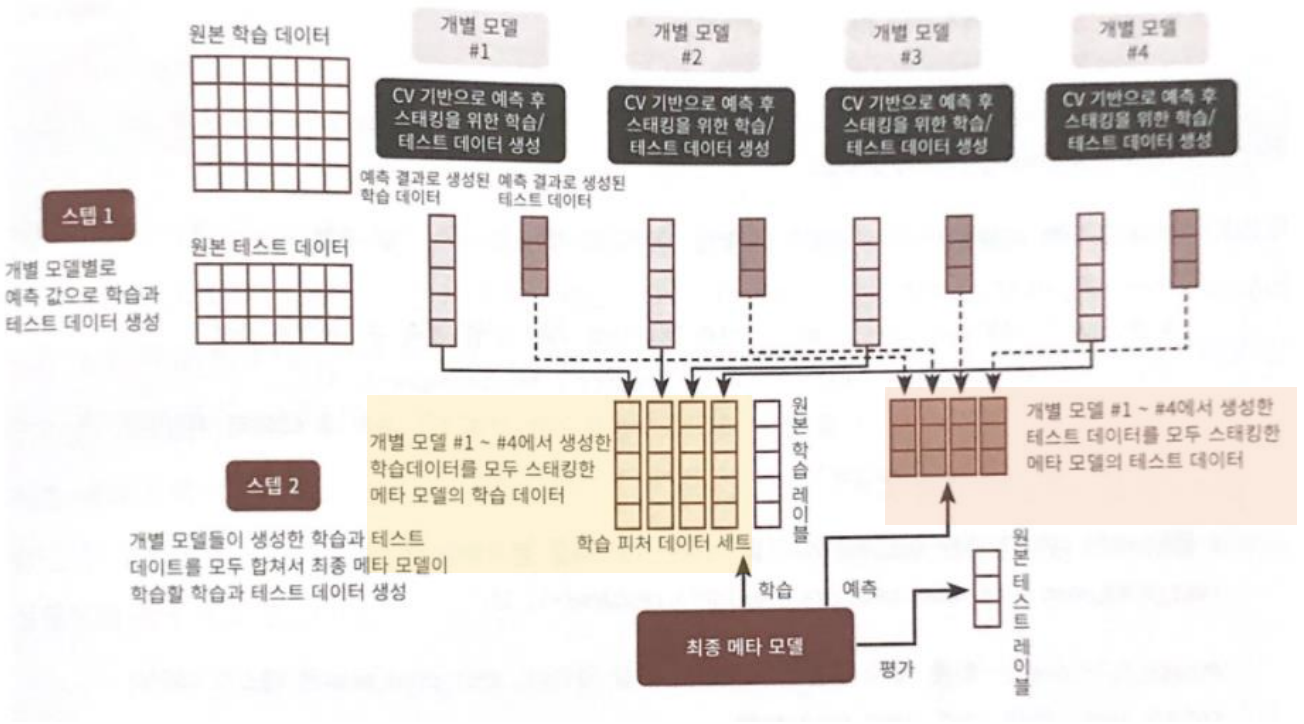
* CV 세트 기반 스태킹의 개념 (STEP 1)



N개의 폴드만큼 반복을 수행하면서 스태킹 데이터를 생성 (N=3)

2. 스태킹 앙상블

* CV 세트 기반 스태킹의 개념 (STEP 2 + 모델 전체)



각 모델들이 STEP 1으로 생성한 학습과 테스트 데이터를 모두 합쳐서 최종적으로
메타 모델이 사용할 학습 데이터와 테스트 데이터를 생성
-> 메타 모델 학습 후 최종 테스트 데이터로 예측, 원본 테스트 레이블 데이터와 비교해 평가

2. 스태킹 앙상블

* CV 세트 기반 스태킹 예제

```
from sklearn.model_selection import KFold
from sklearn.metrics import mean_absolute_error

# 개별 기반 모델에서 최종 메타 모델이 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 생성.
    kf = KFold(n_splits=n_folds, shuffle=False, random_state=0)
    #추후에 메타 모델이 사용할 학습 데이터 반환을 위한 넘파이 배열 초기화
    train_fold_pred = np.zeros((X_train_n.shape[0] ,1 ))
    test_pred = np.zeros((X_test_n.shape[0],n_folds))
    print(model.__class__.__name__ , ' model 시작 ')

    for folder_counter , (train_index, valid_index) in enumerate(kf.split(X_train_n)):
        #입력된 학습 데이터에서 기반 모델이 학습/예측할 폴드 데이터 셋 추출
        print('₩t 폴드 세트: ',folder_counter,' 시작 ')
        X_tr = X_train_n[train_index]
        y_tr = y_train_n[train_index]
        X_te = X_train_n[valid_index]

        #폴드 세트 내부에서 다시 만들어진 학습 데이터로 기반 모델의 학습 수행.
        model.fit(X_tr , y_tr)
        #폴드 세트 내부에서 다시 만들어진 검증 데이터로 기반 모델 예측 후 데이터 저장.
        train_fold_pred[valid_index, :] = model.predict(X_te).reshape(-1,1)
        #입력된 원본 테스트 데이터를 폴드 세트내 학습된 기반 모델에서 예측 후 데이터 저장.
        test_pred[:, folder_counter] = model.predict(X_test_n)

    # 폴드 세트 내에서 원본 테스트 데이터를 예측한 데이터를 평균하여 테스트 데이터로 생성
    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
```

STEP 1

- 메타 모델을 위한 학습/테스트 데이터를 생성하는 get_stacking_base_datasets 함수 정의

- 파라미터 : 개별 모델의 classifier 객체, 원본인 학습용 피처 데이터, 원본인 학습용 레이블 데이터, 원본인 테스트 피처 데이터, k폴드 개수

- 폴드된 학습용 데이터로 학습한 뒤 예측 결과값을 기반으로 메타 모델을 위한 학습/테스트 데이터 생성 (폴드의 개수만큼 반복을 수행)

※ 주의) for문에서 split할때 index가 나눠지면서 연속적인 숫자로 들어가지 않는 경우가 발생하기 때문에 모델을 돌릴때 index를 못찾는 경우 iloc방법을 이용해서 X_tr에 train_index를 넣어줘야함 (복습문제 참고)

2. 스태킹 앙상블

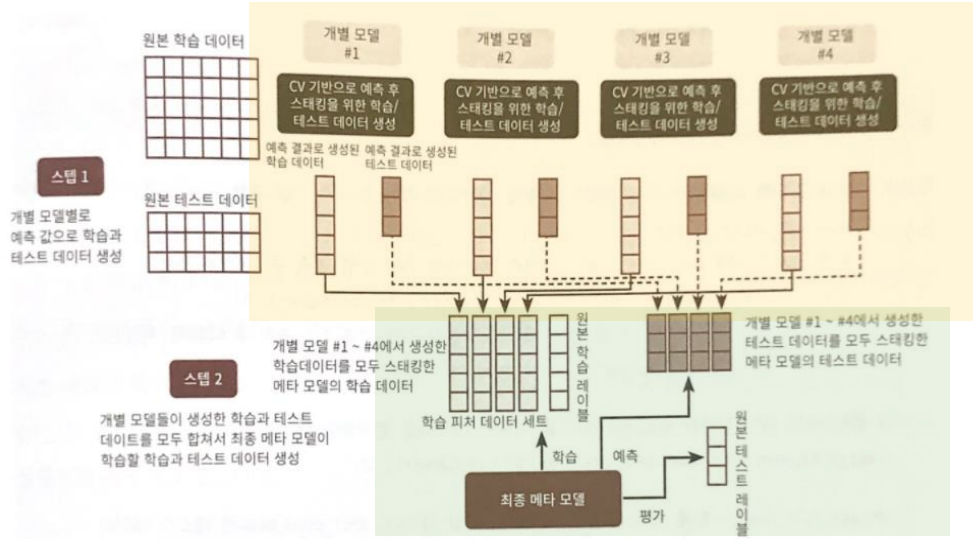
메타 모델이 추후에 사용할 학습용, 테스트용 데이터 세트 반환

```
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)
```

```
KNeighborsClassifier  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
        폴드 세트:  2  시작
        폴드 세트:  3  시작
        폴드 세트:  4  시작
        폴드 세트:  5  시작
        폴드 세트:  6  시작
RandomForestClassifier  model 시작
        폴드 세트:  0  시작
        폴드 세트:  1  시작
```

메타 모델이 학습할 학습용 피처 데이터 세트

메타 모델이 예측할 테스트용 피처 데이터 세트



STEP 2

넘파이의 concatenate() 이용해 각 모델별 학습 데이터와 테스트 데이터 합치기

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

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

```
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)))
```

```
최종 메타 모델의 예측 정확도: 0.9737
```

* 스태킹 모델은 분류뿐만 아니라 회귀에도 적용 가능

3. Auto ML(Automated Machine Learning)

Automated Machine Learning (AutoML)이란?

기계 학습 파이프 라인에서 반복되는 수작업을 자동화하는 프로세스

-> 예측 모델 개발에 많은 시간을 소요했던 코딩, 전처리, 알고리즘 선택, 튜닝 작업을 자동화

PyCaret은 머신러닝 워크 플로우를 자동화하는
Python의 오픈 소스 머신러닝 라이브러리

설치

```
pip install pycaret
```

```
# 버전 확인
import pycaret
pycaret.__version__
```

```
'2.2.0'
```

3. Auto ML(Automated Machine Learning)

데이터 불러오기

```
import pandas as pd
data = pd.read_csv('creditcard.csv')
```

앞서 사용한 creditcard 데이터로 비교/적용해 봅시다.

```
data.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	69.99	0

5 rows × 31 columns

```
# Time열 제거
data.drop('Time',axis=1, inplace=True)
```

3. Auto ML(Automated Machine Learning)

Initialize Setup (초기 세팅)

```
from pycaret.classification import *
```

```
clf = setup(data = data, target = 'Class',train_size=0.8, session_id=6)
```

Random_state와 같은 역할

Processing:

Initiated	01:10:50
Status	Preprocessing Data

Following data types have been inferred automatically, if they are correct press enter to continue or type 'quit' otherwise.

	Data Type
V1	Numeric
V2	Numeric
V3	Numeric
V4	Numeric
V5	Numeric
… …	
V27	Numeric
V28	Numeric
Amount	Numeric
Class	Label

(만약, 'quit' 을 입력한다면 *categorical_feature* 파라미터를 전달하여 type유형을 강제할 수 있다.)

주어진 데이터의 타입을 확인하고

→ 엔터키를 누른다

3. Auto ML(Automated Machine Learning)

Initialize Setup (초기 세팅)

다음과 같이 description이 제공됩니다.

	Description	Value
0	session_id	6
1	Target	Class
2	Target Type	Binary
3	Label Encoded	0: 0, 1: 1
4	Original Data	(284807, 30)
5	Missing Values	False
6	Numeric Features	29
7	Categorical Features	0
8	Ordinal Features	False
9	High Cardinality Features	False
10	High Cardinality Method	None

10	High Cardinality Method	None
11	Transformed Train Set	(227845, 29)
12	Transformed Test Set	(56962, 29)
13	Shuffle Train-Test	True
14	Stratify Train-Test	False
15	Fold Generator	StratifiedKFold
16	Fold Number	10
17	CPU Jobs	-1
18	Use GPU	False
19	Log Experiment	False
20	Experiment Name	clf-default-name

… …

(이하 생략)

3. Auto ML(Automated Machine Learning)

〈참고〉

초기 setup 단계에서, use_gpu=True로 설정하면 GPU를 사용할 수 있다.

15	Fold Generator	StratifiedKFold
16	Fold Number	10
17	CPU Jobs	-1
18	Use GPU	False
19	Log Experiment	False
20	Experiment Name	clf-default-name

→

18	Use GPU	True

use_gpu=True 설정시 True로 바뀜

GPU 지원 알고리즘 :

- Extreme Gradient Boosting
- CatBoost Classifier -> GPU 는 데이터가 50,000 행 이상인 경우에만 활성화 됨
- Light Gradient Boosting Machine -> GPU 설치 필요 https://lightgbm.readthedocs.io/en/latest/GPU-Tutorial.html
- Logistic Regression, Ridge Classifier, Random Forest, K Neighbors Classifier, Support Vector Machine -> cuML> = 0.15 필요 https://github.com/rapidsai/cuml

3. Auto ML(Automated Machine Learning)

```
# GPU가 적용되는 모델 확인
models(internal=True)[['Name', 'GPU Enabled']]
```

use_gpu=False가 디폴트이기 때문에 현재는 모두 False

ID	Name	GPU Enabled
lr	Logistic Regression	False
knn	K Neighbors Classifier	False
nb	Naive Bayes	False
dt	Decision Tree Classifier	False
svm	SVM - Linear Kernel	False
rbfsvm	SVM - Radial Kernel	False
gpc	Gaussian Process Classifier	False
mlp	MLP Classifier	False
ridge	Ridge Classifier	False
rf	Random Forest Classifier	False
qda	Quadratic Discriminant Analysis	False
ada	Ada Boost Classifier	False
gbc	Gradient Boosting Classifier	False
lda	Linear Discriminant Analysis	False
et	Extra Trees Classifier	False
xgboost	Extreme Gradient Boosting	False
lightgbm	Light Gradient Boosting Machine	False
catboost	CatBoost Classifier	False
Bagging	Bagging Classifier	False
Stacking	Stacking Classifier	False
Voting	Voting Classifier	False
CalibratedCV	Calibrated Classifier CV	False

3. Auto ML(Automated Machine Learning)

Compare Baseline (모델 비교)

- 15개 모델 비교
- compare_models()

```
# F1스코어 기준 상위 3개 모델을 best3models에 저장
best3models = compare_models(sort='F1',n_select=3, fold=2)
```

default 는 각각 'Accuracy', 1, 10

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
xgboost	Extreme Gradient Boosting	0.9995	0.9736	0.7843	0.9279	0.8501	0.8498	0.8528	51.4900
catboost	CatBoost Classifier	0.9995	0.9743	0.7817	0.9305	0.8496	0.8494	0.8526	83.1300
rf	Random Forest Classifier	0.9995	0.9447	0.7665	0.9183	0.8354	0.8351	0.8386	66.0550
et	Extra Trees Classifier	0.9995	0.9500	0.7640	0.9154	0.8326	0.8324	0.8359	16.8600
lda	Linear Discriminant Analysis	0.9994	0.9009	0.7614	0.8634	0.8083	0.8079	0.8100	1.8900
svm	SVM - Linear Kernel	0.9993	0.0000	0.7360	0.8231	0.7770	0.7766	0.7779	1.4100
ada	Ada Boost Classifier	0.9992	0.9685	0.6853	0.8165	0.7451	0.7447	0.7476	36.6900
lr	Logistic Regression	0.9992	0.9742	0.6421	0.8753	0.7358	0.7354	0.7467	19.5250
gbc	Gradient Boosting Classifier	0.9992	0.8221	0.6472	0.8343	0.7255	0.7251	0.7327	172.3000
dt	Decision Tree Classifier	0.9990	0.8627	0.7259	0.7073	0.7160	0.7155	0.7158	11.5000
knn	K Neighbors Classifier	0.9991	0.8640	0.5609	0.8911	0.6885	0.6881	0.7066	142.1100
ridge	Ridge Classifier	0.9989	0.0000	0.4264	0.8526	0.5663	0.5658	0.6012	0.9300
lightgbm	Light Gradient Boosting Machine	0.9951	0.6921	0.5406	0.1857	0.2764	0.2745	0.3149	6.4950
nb	Naive Bayes	0.9778	0.9623	0.8325	0.0618	0.1150	0.1121	0.2232	0.9200
qda	Quadratic Discriminant Analysis	0.9768	0.9684	0.8629	0.0611	0.1140	0.1112	0.2260	1.2400

(참고) 현재 로컬 환경
노트북 사양(문서용)
- CPU: 8세대 i3(4C4T)
- RAM: 8GB 기준
약 24분정도 소요됩니다.

default가 n_jobs=-1이기 때문에,
컴퓨터의 모든 코어를 사용하여
코드 돌리는 동안 속도가 느려집니다.

3. Auto ML(Automated Machine Learning)

Create Model (모델 만들기)

- 로지스틱 회귀
- create_model()

```
# 로지스틱 회귀
lr = create_model('lr', fold = 2)
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9993	0.9838	0.7360	0.8580	0.7923	0.7920	0.7944
1	0.9991	0.9646	0.5482	0.8926	0.6792	0.6788	0.6991
Mean	0.9992	0.9742	0.6421	0.8753	0.7358	0.7354	0.7467
SD	0.0001	0.0096	0.0939	0.0173	0.0566	0.0566	0.0476

```
lr
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=6, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

3. Auto ML(Automated Machine Learning)

Create Model (모델 만들기)

- LightGBM
- create_model()

```
# lightgbm
lgbm = create_model('lightgbm', fold= 2 )
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9953	0.6736	0.5178	0.1865	0.2742	0.2723	0.3089
1	0.9950	0.7106	0.5635	0.1850	0.2785	0.2767	0.3210
Mean	0.9951	0.6921	0.5406	0.1857	0.2764	0.2745	0.3149
SD	0.0002	0.0185	0.0228	0.0007	0.0022	0.0022	0.0061

```
lgbm
```

```
LGBMClassifier(boosting_type='gbdt', class_weight=None, colsample_bytree=1.0,
               importance_type='split', learning_rate=0.1, max_depth=-1,
               min_child_samples=20, min_child_weight=0.001, min_split_gain=0.0,
               n_estimators=100, n_jobs=-1, num_leaves=31, objective=None,
               random_state=6, reg_alpha=0.0, reg_lambda=0.0, silent=True,
               subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
```

3. Auto ML(Automated Machine Learning)

Tune Hyperparameters(하이퍼 파라미터 튜닝)

- 로지스틱 회귀
- tune_model()

```
# 로지스틱 회귀
tuned_lr = tune_model(lr, fold = 2, optimize='F1')
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9993	0.9839	0.7411	0.8588	0.7956	0.7953	0.7975
1	0.9991	0.9633	0.5584	0.8943	0.6875	0.6871	0.7063
Mean	0.9992	0.9736	0.6497	0.8766	0.7416	0.7412	0.7519
SD	0.0001	0.0103	0.0914	0.0177	0.0541	0.0541	0.0456

```
tuned_lr
```

```
LogisticRegression(C=8.026, class_weight={}, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=6, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

3. Auto ML(Automated Machine Learning)

Tune Hyperparameters(하이퍼 파라미터 튜닝)

- LightGBM
- tune_model()

```
# lightgbm
tuned_lgbm = tune_model(lgbm, fold = 2, optimize='F1')
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9995	0.9786	0.7868	0.9226	0.8493	0.8491	0.8518
1	0.9995	0.9805	0.7411	0.9419	0.8295	0.8293	0.8353
Mean	0.9995	0.9795	0.7640	0.9323	0.8394	0.8392	0.8435
SD	0.0000	0.0009	0.0228	0.0097	0.0099	0.0099	0.0083

```
tuned_lgbm
```

```
LGBMClassifier(bagging_fraction=0.5, bagging_freq=2, boosting_type='gbdt',
               class_weight=None, colsample_bytree=1.0, feature_fraction=0.9,
               importance_type='split', learning_rate=0.234, max_depth=-1,
               min_child_samples=75, min_child_weight=0.001, min_split_gain=0.3,
               n_estimators=100, n_jobs=-1, num_leaves=80, objective=None,
               random_state=6, reg_alpha=0.5, reg_lambda=2, silent=True,
               subsample=1.0, subsample_for_bin=200000, subsample_freq=0)
```

3. Auto ML(Automated Machine Learning)

하이퍼 파라미터 튜닝 결과 비교

로지스틱 회귀

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9993	0.9838	0.7360	0.8580	0.7923	0.7920	0.7944
1	0.9991	0.9646	0.5482	0.8926	0.6792	0.6788	0.6991
Mean	0.9992	0.9742	0.6421	0.8753	0.7358	0.7354	0.7467
SD	0.0001	0.0096	0.0939	0.0173	0.0566	0.0566	0.0476

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9993	0.9839	0.7411	0.8588	0.7956	0.7953	0.7975
1	0.9991	0.9633	0.5584	0.8943	0.6875	0.6871	0.7063
Mean	0.9992	0.9736	0.6497	0.8766	0.7416	0.7412	0.7519
SD	0.0001	0.0103	0.0914	0.0177	0.0541	0.0541	0.0456

소폭 상승

LightGBM

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9953	0.6736	0.5178	0.1865	0.2742	0.2723	0.3089
1	0.9950	0.7106	0.5635	0.1850	0.2785	0.2767	0.3210
Mean	0.9951	0.6921	0.5406	0.1857	0.2764	0.2745	0.3149
SD	0.0002	0.0185	0.0228	0.0007	0.0022	0.0022	0.0061

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9995	0.9786	0.7868	0.9226	0.8493	0.8491	0.8518
1	0.9995	0.9805	0.7411	0.9419	0.8295	0.8293	0.8353
Mean	0.9995	0.9795	0.7640	0.9323	0.8394	0.8392	0.8435
SD	0.0000	0.0009	0.0228	0.0097	0.0099	0.0099	0.0083

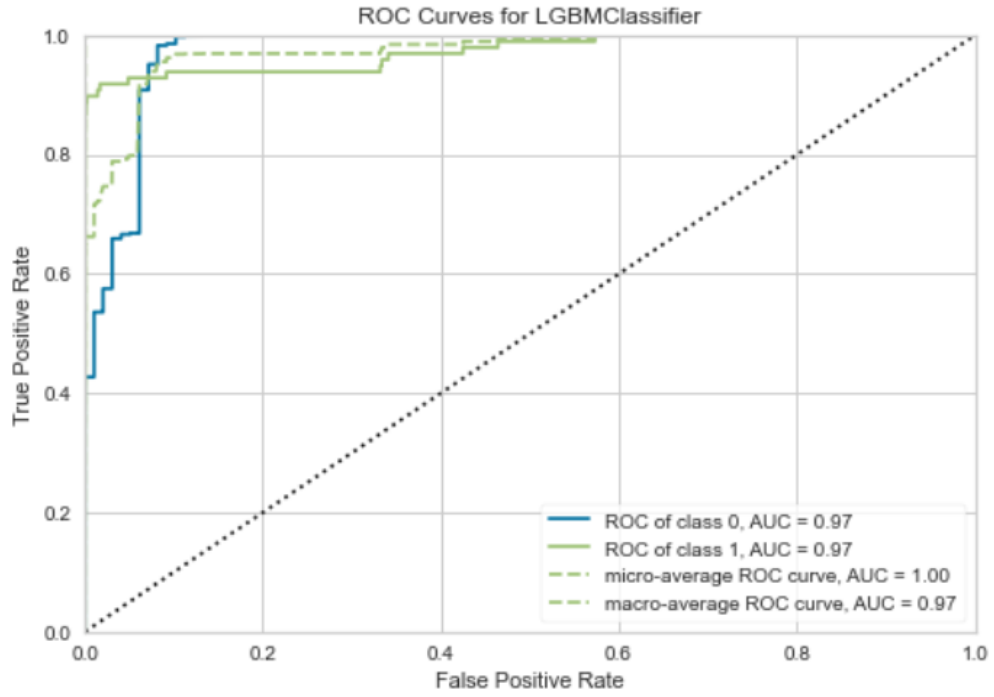
대폭 상승

3. Auto ML(Automated Machine Learning)

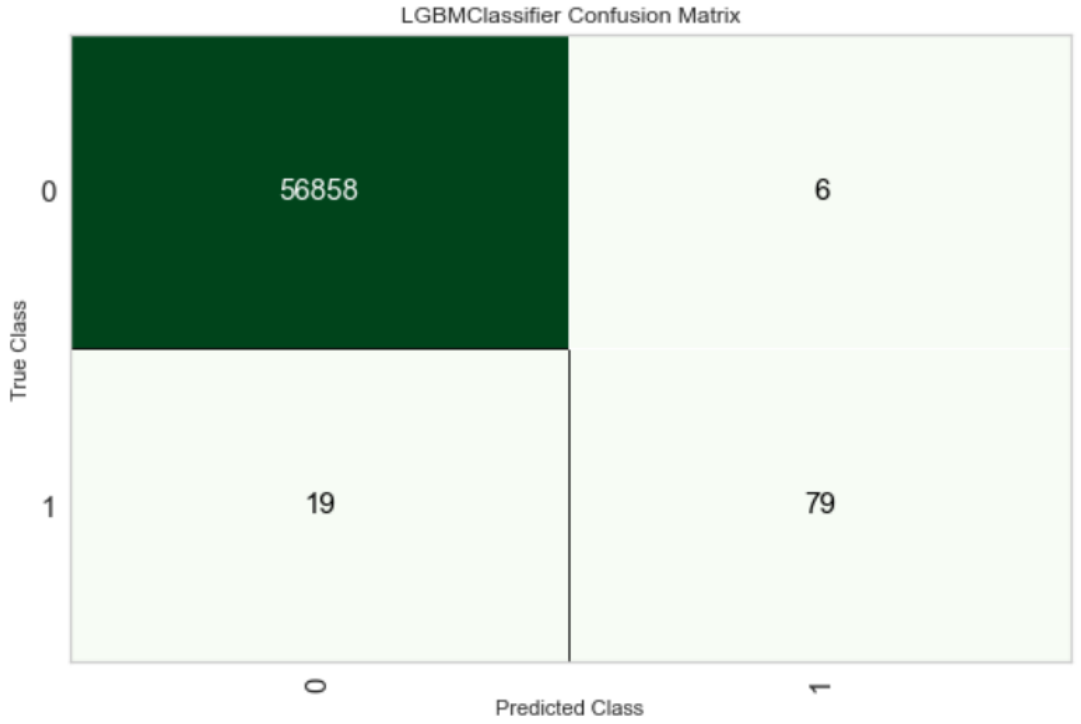
Analyze Model (다양한 시각화)

- plot_model()

```
plot_model(tuned_lgbm, 'auc')
```




```
plot_model(tuned_lgbm,'confusion_matrix')
```

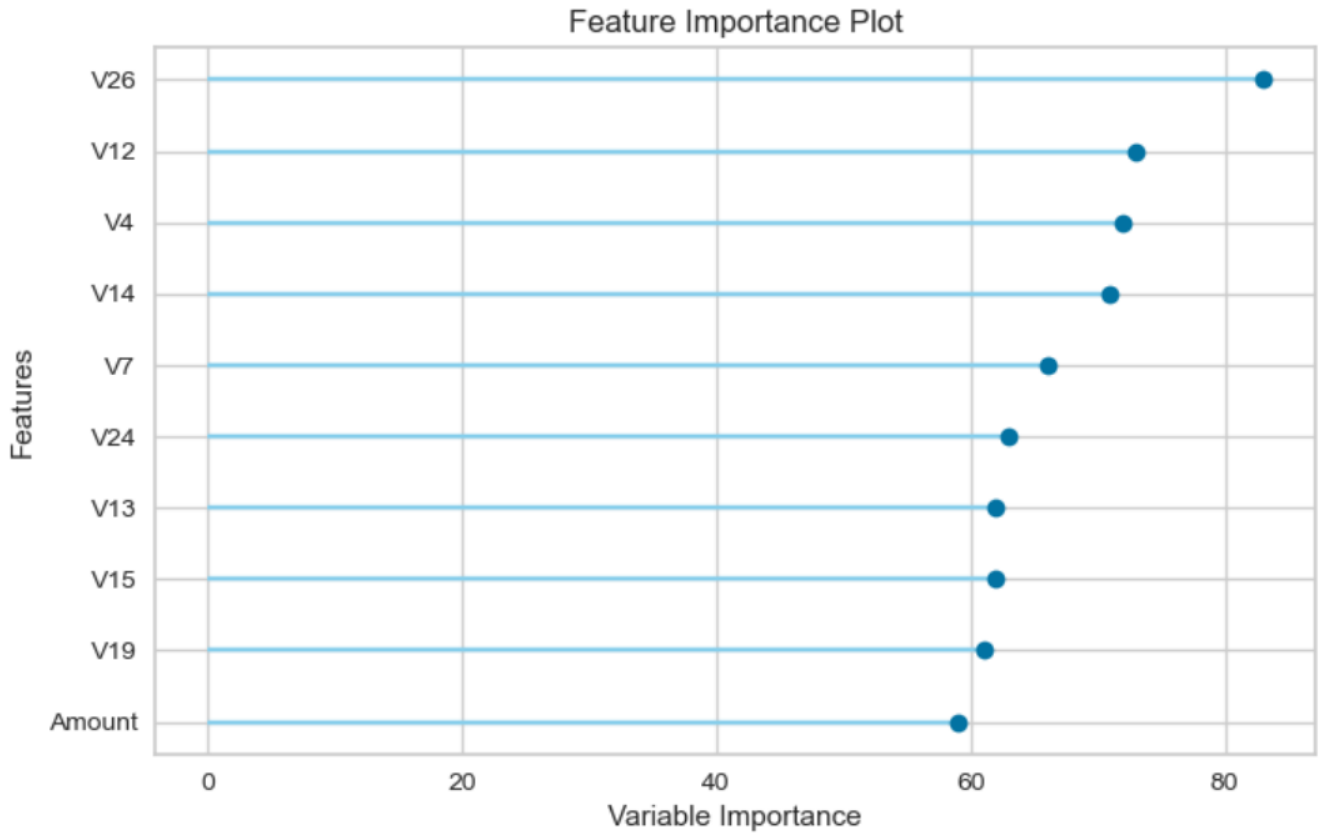


3. Auto ML(Automated Machine Learning)

Analyze Model (다양한 시각화)

- plot_model()

```
plot_model(tuned_lgbm, plot = 'feature')
```

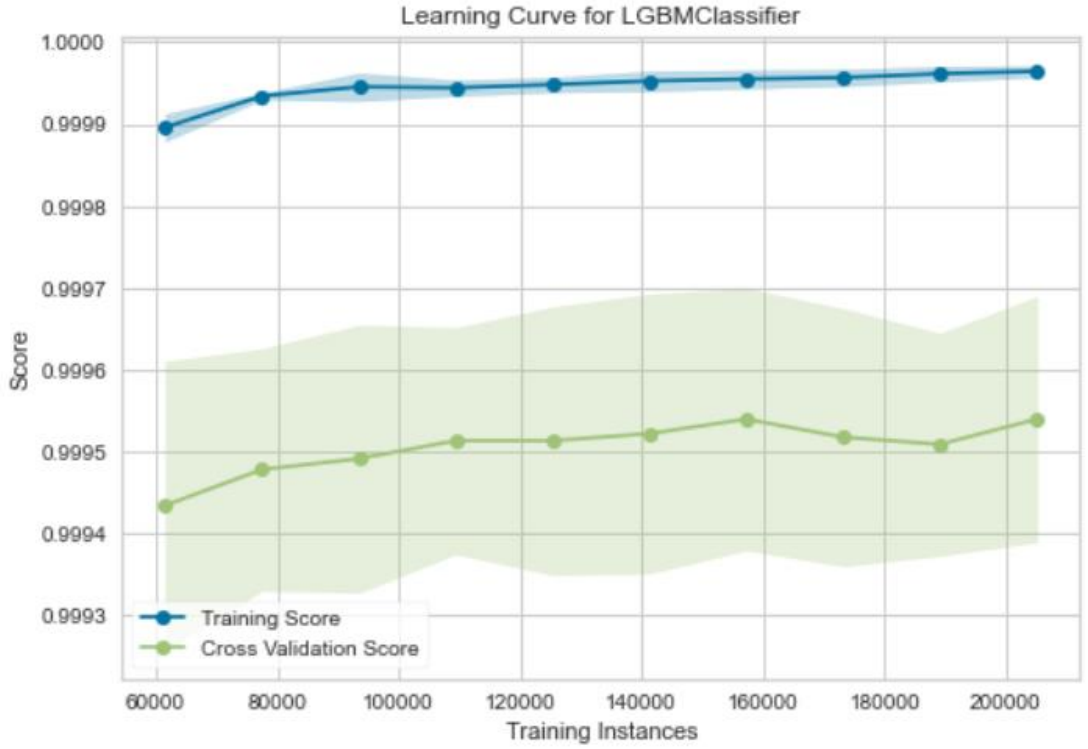


3. Auto ML(Automated Machine Learning)

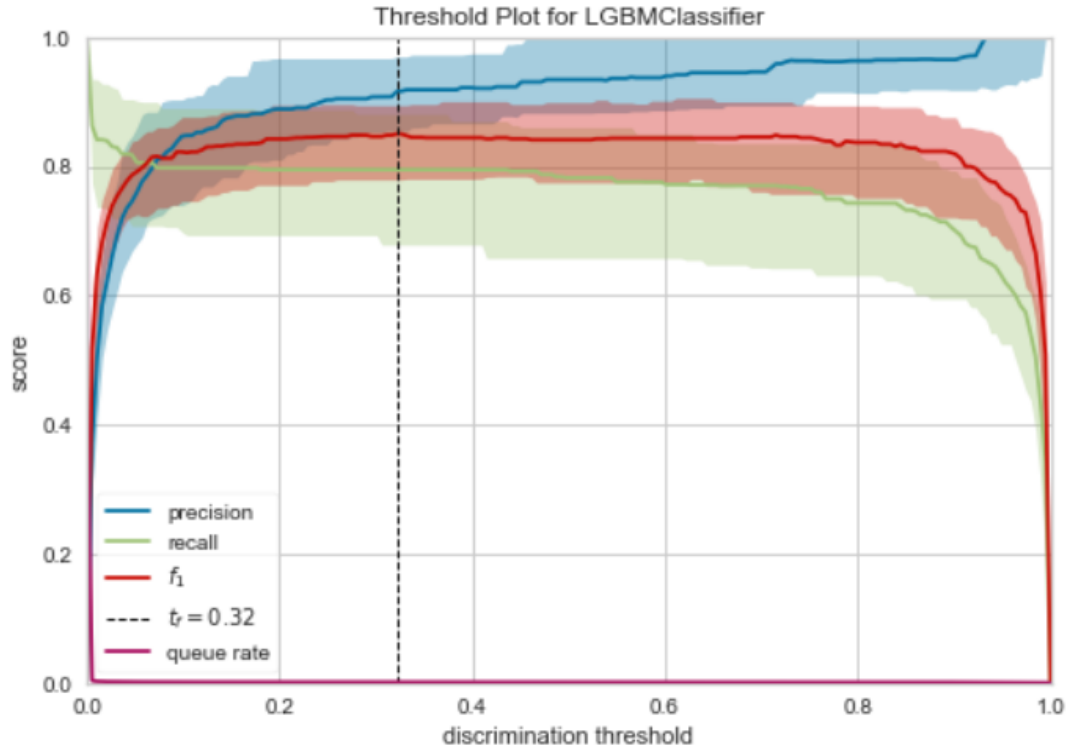
Analyze Model (다양한 시각화)

- plot_model()

```
plot_model(tuned_lgbm,'learning') # 약 4분 소요됩니다.
```




```
plot_model(tuned_lgbm, 'threshold') # 약 4분 소요됩니다.
```



3. Auto ML(Automated Machine Learning)

Blend Models (모델 결합)

- 다시 돌아와서, 앞서 만든 best3models의 세 모델로 앙상블(soft vote)하기
- blend_models()

```
blended = blend_models(estimator_list = best3models, fold = 2, method = 'soft') # 약 15분 소요됩니다.
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.9996	0.9824	0.7919	0.9455	0.8619	0.8617	0.8651
1	0.9995	0.9783	0.7665	0.9321	0.8412	0.8410	0.8450
Mean	0.9995	0.9804	0.7792	0.9388	0.8516	0.8513	0.8550
SD	0.0000	0.0021	0.0127	0.0067	0.0103	0.0103	0.0100

create_model()로 직접 만든 모델을 사용해도 됨
예를들어, estimator_list=[tuned_lr, tuned_lgbm]

Default는 'auto'
'auto'는 'soft'를 사용하고,
'soft'가 지원되지 않는 경우 'hard'로 대체됨

3. Auto ML(Automated Machine Learning)

모델 예측 (Prediction)

- predict_model()
- setup 환경에 이미 hold-out set이 존재함

```
pred_holdout = predict_model(blended)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	Voting Classifier	0.9996	0.9787	0.8061	0.9634	0.8778	0.8776	0.8811

하이퍼 파라미터 튜닝을 하지않은
상위 3개의 모델을 사용했는데도 성능이 꽤 좋다!

* 참고
정확히는 finalize_model() 함수를 사용하여 전체 데이터에 대해 학습을 하고, predict_model()을 이용하여 실제 test 데이터에 적용을 해야합니다.

결론

- 정규화, 로그변환, 이상치제거, SMOTE 등을 직접 하지 않고 모두 기계에게 맡기는 AutoML
- Pycaret 이 제공하는 기본적인 기능을 사용했으나, 유의미한 결과를 얻을 수 있었음
- 추가 작업을 통해 성능을 더 향상시킬 수 있을거라 기대됨
- 홈페이지와 공식 문서가 매우 잘 정리되어 있으니 관심있다면 공부해 보는 것도 좋을 것 같습니다.

https://pycaret.org/
https://pycaret.readthedocs.io/en/latest/index.html#

4. 성인 인구 소득 예측 예제

1994년 미국 성인을 대상으로 한 데이터

출처: https://www.kaggle.com/c/kakr-4th-competition/overview

목표: F1 스코어 높이기

* 변수설명

- id
- age : 나이
- workclass : 고용 형태
- fnlwgt : 사람 대표성을 나타내는 가중치 (final weight의 약자)
- education : 교육 수준
- education_num : 교육 수준 수치
- marital_status: 결혼 상태
- occupation : 업종
- relationship : 가족 관계
- race : 인종
- sex : 성별
- capital_gain : 양도 소득
- capital_loss : 양도 손실
- hours_per_week : 주당 근무 시간
- native_country : 국적
- income : 수익 (예측해야 하는 값)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import os

warnings.filterwarnings('ignore')

train = pd.read_csv('/train.csv')
test = pd.read_csv('/python/test.csv')
```

```
print(train.shape, test.shape)
```

```
(26049, 16) (6512, 15)
```

train, test 의 분포를 따로 봐야하는 이유는 캐글에서 가끔 함정으로 train 에 없는 값 분포를 test에 심어 놓기 때문

4. 성인 인구 소득 예측 예제

```
train.head(3)
```

	id	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_we
0	0	40	Private	168538	HS-grad	9	Married-civ-spouse	Sales	Husband	White	Male	0	0	
1	1	17	Private	101626	9th	5	Never-married	Machine-op-inspct	Own-child	White	Male	0	0	
2	2	18	Private	353358	Some-college	10	Never-married	Other-service	Own-child	White	Male	0	0	

```
test.head(3)
```

	id	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_
0	0	28	Private	67661	Some-college	10	Never-married	Adm-clerical	Other-relative	White	Female	0	0	
1	1	40	Self-emp-inc	37869	HS-grad	9	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	
2	2	20	Private	109952	Some-college	10	Never-married	Handlers-cleaners	Own-child	White	Male	0	0	

4. 성인 인구 소득 예측 예제

```
# 결측치
print(train.isnull().sum(),'₩n')
print(test.isnull().sum())
```

```
id                0
age               0
workclass         0
fnlwgt            0
education         0
education_num     0
marital_status    0
occupation        0
relationship      0
race              0
sex               0
capital_gain      0
capital_loss      0
hours_per_week    0
native_country    0
income            0
dtype: int64
```

```
id                0
age               0
workclass         0
fnlwgt            0
education         0
education_num     0
marital_status    0
occupation        0
relationship      0
race              0
sex               0
capital_gain      0
capital_loss      0
hours_per_week    0
native_country    0
dtype: int64
```

결측치가 존재하지 않는다

```
# 컬럼 별 info() 확인
print(train.info(),'₩n')
print(test.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26049 entries, 0 to 26048
Data columns (total 16 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              26049 non-null  int64
 1   age             26049 non-null  int64
 2   workclass       26049 non-null  object
 3   fnlwgt          26049 non-null  int64
 4   education       26049 non-null  object
 5   education_num   26049 non-null  int64
 6   marital_status  26049 non-null  object
 7   occupation      26049 non-null  object
 8   relationship    26049 non-null  object
 9   race            26049 non-null  object
 10  sex             26049 non-null  object
 11  capital_gain    26049 non-null  int64
 12  capital_loss    26049 non-null  int64
 13  hours_per_week  26049 non-null  int64
 14  native_country  26049 non-null  object
 15  income          26049 non-null  object
dtypes: int64(7), object(9)
memory usage: 3.2+ MB
None
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6512 entries, 0 to 6511
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              6512 non-null   int64
 1   age             6512 non-null   int64
 2   workclass       6512 non-null   object
 3   fnlwgt          6512 non-null   int64
 4   education       6512 non-null   object
 5   education_num   6512 non-null   int64
 6   marital_status  6512 non-null   object
 7   occupation      6512 non-null   object
 8   relationship    6512 non-null   object
 9   race            6512 non-null   object
 10  sex             6512 non-null   object
 11  capital_gain    6512 non-null   int64
 12  capital_loss    6512 non-null   int64
 13  hours_per_week  6512 non-null   int64
 14  native_country  6512 non-null   object
dtypes: int64(7), object(8)
memory usage: 763.2+ KB
None
```

4. 성인 인구 소득 예측 예제

```
# Target 변환 (income)
train['income'].value_counts()
```

```
<=50K    19744
>50K      6305
Name: income, dtype: int64
```

```
# 소득이 50k이하인 사람은 0, 그렇지 않으면 1 (즉, 저소득자 0, 고소득자 1)
train['income'] = train['income'].apply(lambda x: 0 if x == '<=50K' else 1)
train['income'].value_counts()
```

```
0    19744
1     6305
Name: income, dtype: int64
```

```
all_data = pd.concat([train, test])
all_data.head(1)
```

train, test 데이터 확인 결과 별다른 이상이 없으므로 train+test로 병합하여 전처리한다.

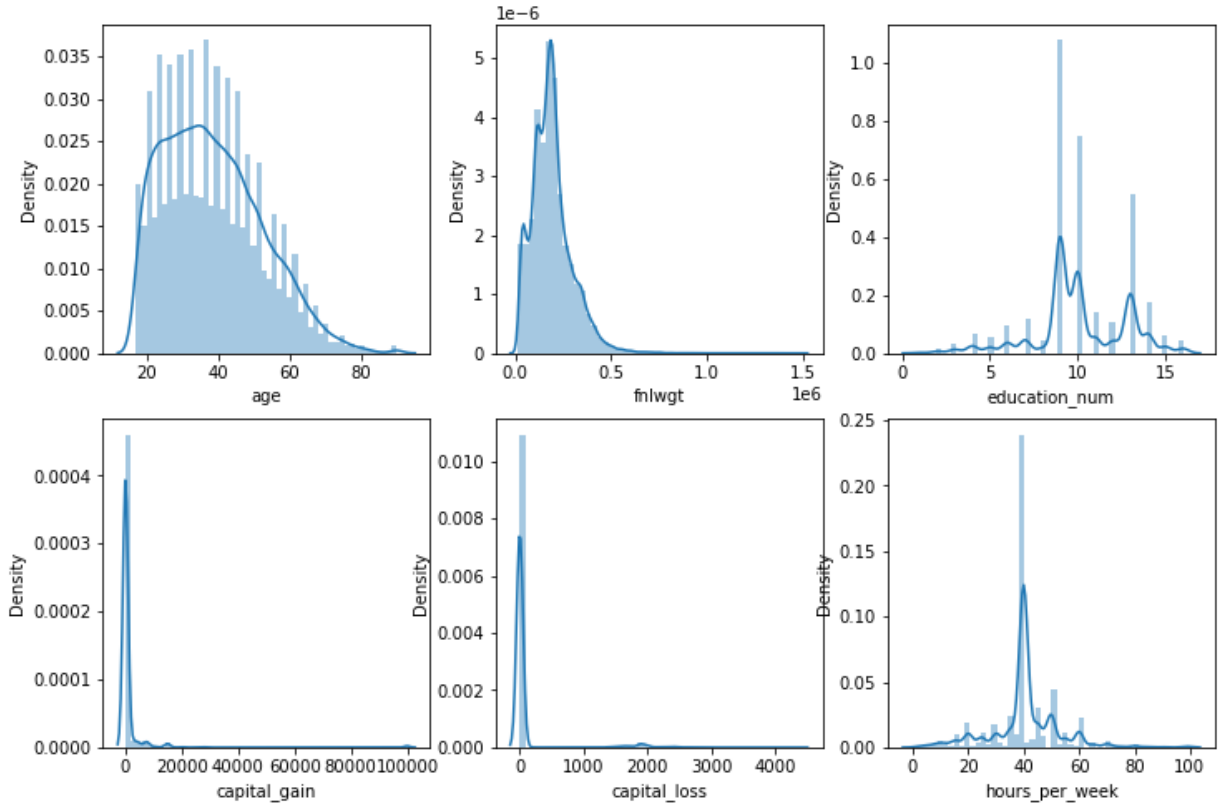
	id	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_w
0	0	40	Private	168538	HS-grad	9	Married-civ-spouse	Sales	Husband	White	Male	0	0	

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (int64형)

- age : 나이
- fnlwgt : 사람 대표성을 나타내는 가중치 (final weight의 약자)
- education_num : 교육 수준
- capital_gain : 양도 소득
- capital_loss : 양도 손실
- hours_per_week : 주당 근무 시간

```
plt.figure(figsize=(12,8))
plt.subplot(231)
sns.distplot(all_data['age'])
plt.subplot(232)
sns.distplot(all_data['fnlwgt'])
plt.subplot(233)
sns.distplot(all_data['education_num'])
plt.subplot(234)
sns.distplot(all_data['capital_gain'])
plt.subplot(235)
sns.distplot(all_data['capital_loss'])
plt.subplot(236)
sns.distplot(all_data['hours_per_week'])
```

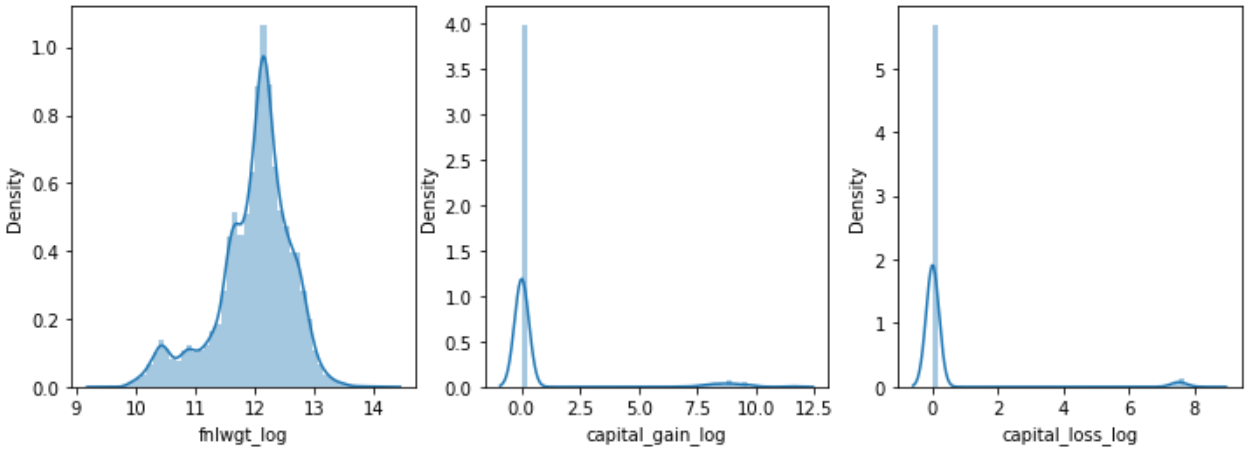


fnlwgt, capital_grain, capital_loss의 분포가 치우쳐있다. -> 로그변환!

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (int64형)

```
# fnlwgt, capital_gain, capital_loss에 로그를 취한다.
all_data['fnlwgt_log'] = np.log1p(all_data['fnlwgt'])
all_data['capital_gain_log'] = np.log1p(all_data['capital_gain'])
all_data['capital_loss_log'] = np.log1p(all_data['capital_loss'])

plt.figure(figsize=(12,4))
plt.subplot(131)
sns.distplot(all_data['fnlwgt_log'])
plt.subplot(132)
sns.distplot(all_data['capital_gain_log'])
plt.subplot(133)
sns.distplot(all_data['capital_loss_log'])
```



capital_gain과 capital_loss에는 큰 변화가 없음

4. 성인 인구 소득 예측 예제

- education_num(교육 수준 수치)

```
# 비슷한 이름을 가진 education 칼럼과 비교
print(all_data['education'].value_counts(),'₩n')
print(all_data['education_num'].value_counts())
```

```
# 두 칼럼이 동일하기 때문에 둘 중 하나는 제거
all_data = all_data.drop('education_num', axis=1)
```

```
HS-grad         10501
Some-college     7291
Bachelors        5355
Masters          1723
Assoc-voc        1382
11th             1175
Assoc-acdm       1067
10th              933
7th-8th           646
Prof-school       576
9th               514
12th              433
Doctorate         413
5th-6th           333
1st-4th           168
Preschool          51
Name: education, dtype: int64
```

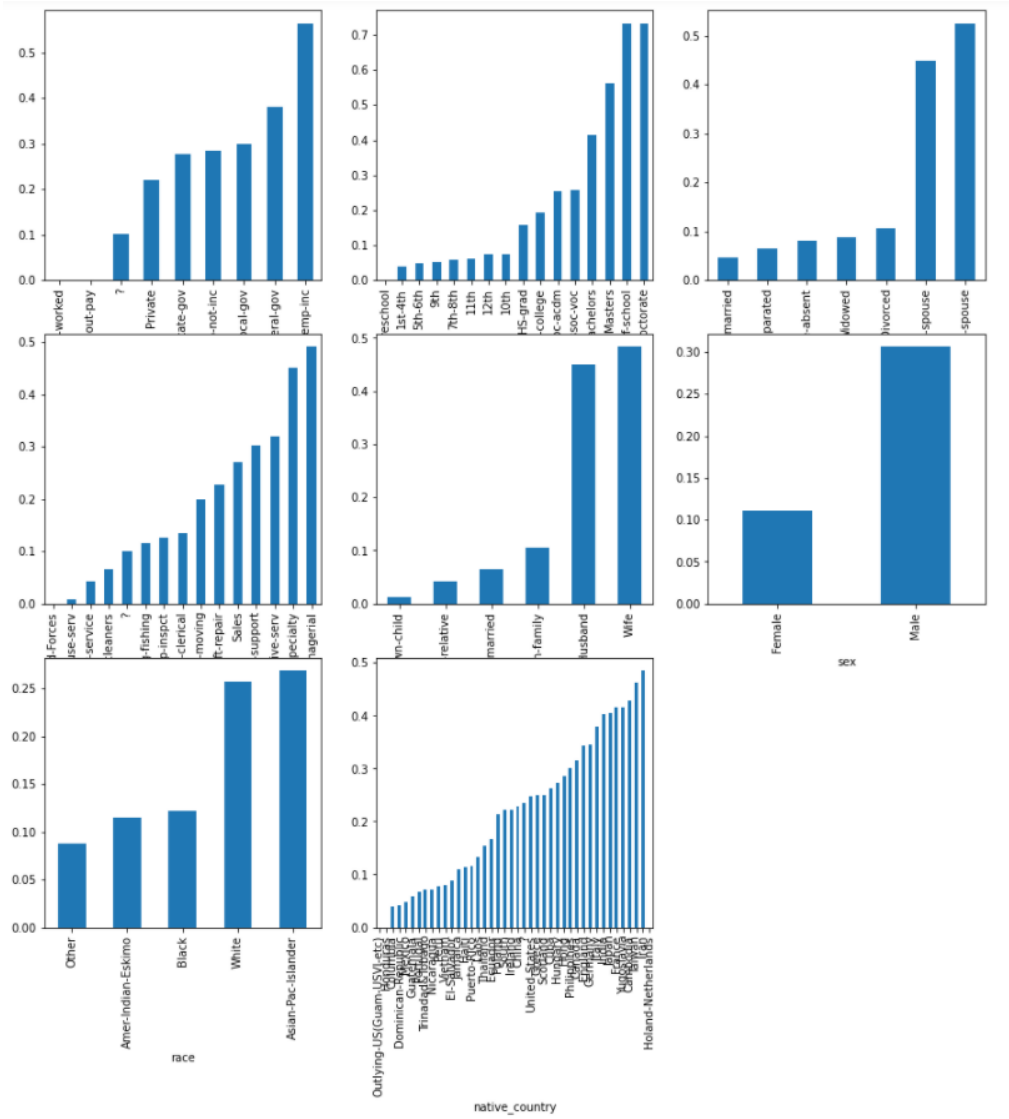
```
9     10501
10     7291
13     5355
14     1723
11     1382
7      1175
12     1067
6       933
4       646
15      576
5       514
8       433
16      413
3       333
2       168
1        51
Name: education_num, dtype: int64
```

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- workclass : 고용 형태
- education : 교육 수준
- marital_status: 결혼 상태
- occupation : 업종
- relationship : 가족 관계
- race : 인종
- sex : 성별
- native_country : 국적

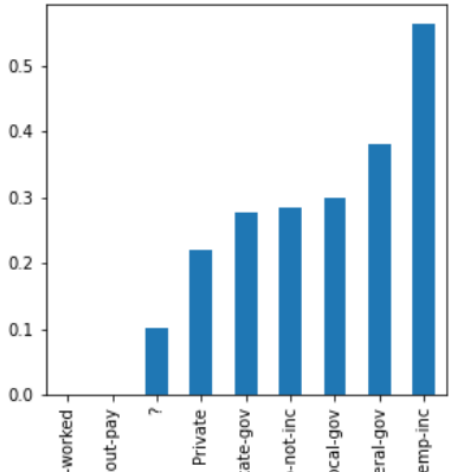
```
plt.figure(figsize=(15,15))
plt.subplot(331)
all_data.groupby('workclass')['income'].mean().sort_values().plot(kind='bar')
plt.subplot(332)
all_data.groupby('education')['income'].mean().sort_values().plot(kind='bar')
plt.subplot(333)
all_data.groupby('marital_status')['income'].mean().sort_values().plot(kind='bar')
plt.subplot(334)
all_data.groupby('occupation')['income'].mean().sort_values().plot(kind='bar')
plt.subplot(335)
all_data.groupby('relationship')['income'].mean().sort_values().plot(kind='bar')
plt.subplot(336)
all_data.groupby('sex')['income'].mean().sort_values().plot(kind='bar' )
plt.subplot(337)
all_data.groupby('race')['income'].mean().sort_values().plot(kind='bar')
plt.subplot(338)
all_data.groupby('native_country')['income'].mean().sort_values().plot(kind='bar')
```



4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- workclass(고용 형태)



Never-worked와 Without-pay의 평균 income이 0이다.

```
all_data.groupby('workclass')['income'].sum()
```

```
workclass
?                  153.0
Federal-gov        292.0
Local-gov          505.0
Never-worked         0.0
Private           3993.0
Self-emp-inc       496.0
Self-emp-not-inc   577.0
State-gov          289.0
Without-pay          0.0
Name: income, dtype: float64
```

Sum으로 확인한 결과 0

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

```
Private             22696
Self-emp-not-inc     2541
Local-gov            2093
?                    1836
State-gov            1298
Self-emp-inc         1116
Federal-gov           960
Without-pay            14
Never-worked            7
Name: workclass, dtype: int64
```

```
# Never-worked와 Without-pay를 Other로 하나로 합친다.
workclass_other = ['Without-pay', 'Never-worked']
all_data['workclass'] = all_data['workclass'].apply(lambda x: 'Other' if x in workclass_other else x)
```

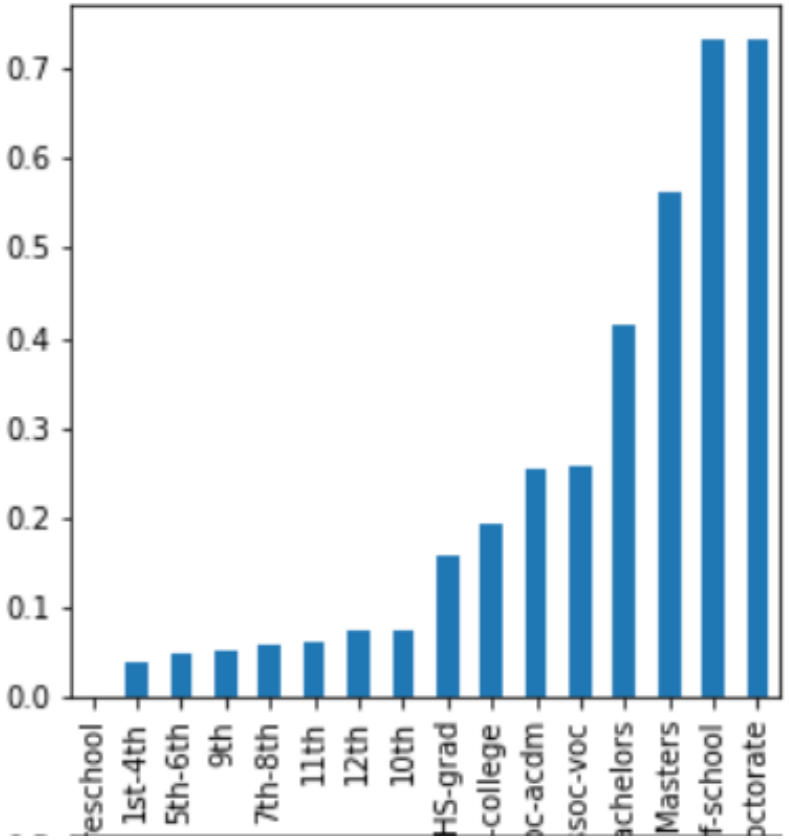
```
all_data['workclass'].value_counts()
```

```
Private             22696
Self-emp-not-inc     2541
Local-gov            2093
?                    1836
State-gov            1298
Self-emp-inc         1116
Federal-gov           960
Other                  21
Name: workclass, dtype: int64
```

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- education(교육 수준)




```
all_data.groupby(['education'])['income'].agg(['mean', 'count']).sort_values('mean')
```

education	mean	count
Preschool	0.000000	40
1st-4th	0.037313	134
5th-6th	0.049057	265
9th	0.052632	418
7th-8th	0.057426	505
11th	0.059653	922
12th	0.072423	359
10th	0.072503	731
HS-grad	0.158544	8433
Some-college	0.192586	5800
Assoc-acdm	0.255344	842
Assoc-voc	0.255474	1096
Bachelors	0.415516	4344
Masters	0.561684	1378
Prof-school	0.733906	466
Doctorate	0.734177	316

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- education(교육 수준)

단계가 너무 많으면 모델 학습시 과적합이 일어날 수 있으므로, 8단계로 소득분위 지정

```
# 8단계로 소득 분위 임의 지정
education_map = {
    'Preschool': 'level_0',
    '1st-4th': 'level_1',
    '5th-6th': 'level_1',
    '7th-8th': 'level_1',
    '9th': 'level_2',
    '10th': 'level_2',
    '11th': 'level_2',
    '12th': 'level_2',
    'HS-grad': 'level_3',
    'Some-college': 'level_3',
    'Assoc-acdm': 'level_4',
    'Assoc-voc': 'level_4',
    'Bachelors': 'level_5',
    'Masters': 'level_6',
    'Prof-school': 'level_7',
    'Doctorate': 'level_7',
}
```

```
all_data['education'] = all_data['education'].map(education_map)
```

복사+붙여넣기

```
# 8단계로 소득 분위 임의 지정
education_map = {
    'Preschool': 'level_0',
    '1st-4th': 'level_1',
    '5th-6th': 'level_1',
    '7th-8th': 'level_1',
    '9th': 'level_2',
    '10th': 'level_2',
    '11th': 'level_2',
    '12th': 'level_2',
    'HS-grad': 'level_3',
    'Some-college': 'level_3',
    'Assoc-acdm': 'level_4',
    'Assoc-voc': 'level_4',
    'Bachelors': 'level_5',
    'Masters': 'level_6',
    'Prof-school': 'level_7',
    'Doctorate': 'level_7',
}
```

4. 성인 인구 소득 예측 예제

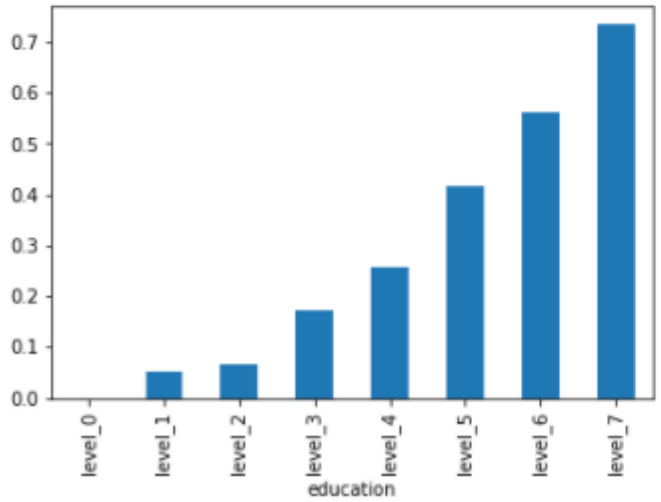
- 칼럼별로 살펴보기 (object64형)
  - education(교육 수준)

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

```
level_3    17792
level_5     5355
level_2     3055
level_4     2449
level_6     1723
level_1     1147
level_7      989
level_0       51
Name: education, dtype: int64
```

```
all_data.groupby('education')['income'].mean().sort_values().plot(kind='bar')
```

```
<AxesSubplot:xlabel='education'>
```



4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- occupation(업종)

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

```
Prof-specialty       4140
Craft-repair         4099
Exec-managerial      4066
Adm-clerical         3770
Sales                3650
Other-service        3295
Machine-op-inspct    2002
?                    1843
Transport-moving     1597
Handlers-cleaners    1370
Farming-fishing       994
Tech-support          928
Protective-serv       649
Priv-house-serv       149
Armed-Forces            9
Name: occupation, dtype: int64
```

```
all_data.groupby('occupation')['income'].sum()
```

```
occupation
?                    153.0
Adm-clerical         402.0
Armed-Forces           0.0
Craft-repair         738.0
Exec-managerial     1593.0
Farming-fishing       91.0
Handlers-cleaners     71.0
Machine-op-inspct    200.0
Other-service        110.0
Priv-house-serv        1.0
Prof-specialty      1491.0
Protective-serv      167.0
Sales                811.0
Tech-support         224.0
Transport-moving     253.0
Name: income, dtype: float64
```

Armed-Forces가 매우 소수이고, Armed-Forces의 income은 모두 0임을 알 수 있다.
-> 과적합 방지를 위해 Armed-Forces을 Priv-house-serve에 포함

```
# 과적합 방지를 위해 Armed-Forces을 Priv-house-serve에 포함시킨다.
all_data.loc[all_data['occupation'].isin(['Armed-Forces', 'Priv-house-serv']), 'occupation'] = 'Priv-house-serv'
```

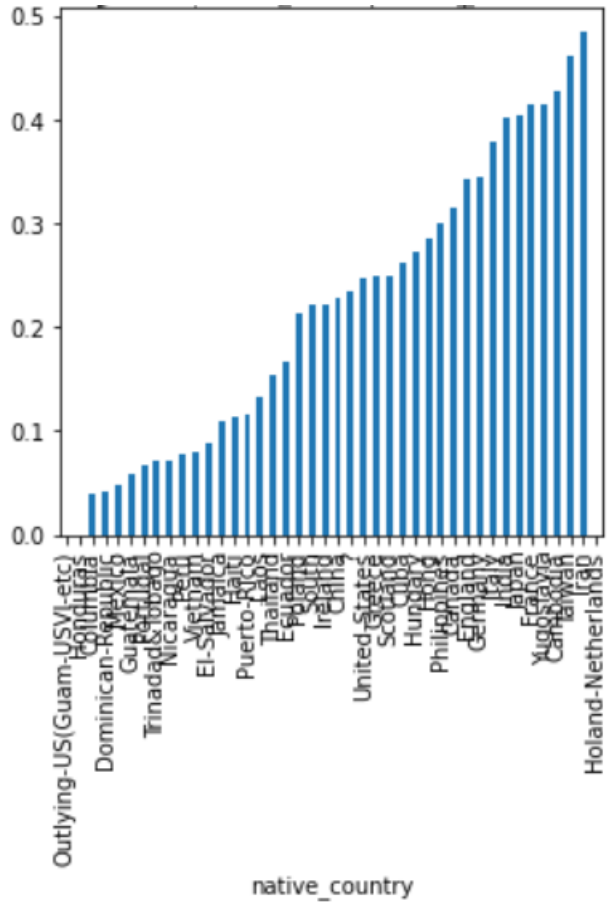
```
all_data['occupation'].value_counts()
```

```
Prof-specialty       4140
Craft-repair         4099
Exec-managerial      4066
Adm-clerical         3770
Sales                3650
Other-service        3295
Machine-op-inspct    2002
?                    1843
Transport-moving     1597
Handlers-cleaners    1370
Farming-fishing       994
Tech-support          928
Protective-serv       649
Priv-house-serv       158
Name: occupation, dtype: int64
```

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- native_country(국적)




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

United-States	29170	Haiti	44
Mexico	643	Iran	43
?	583	Portugal	37
Philippines	198	Nicaragua	34
Germany	137	Peru	31
Canada	121	Greece	29
Puerto-Rico	114	France	29
El-Salvador	106	Ecuador	28
India	100	Ireland	24
Cuba	95	Hong	20
England	90	Trinadad&Tobago	19
Jamaica	81	Cambodia	19
South	80	Thailand	18
China	75	Laos	18
Italy	73	Yugoslavia	16
Dominican-Republic	70	Outlying-US(Guam-USVI-etc)	14
Vietnam	67	Honduras	13
Guatemala	64	Hungary	13
Japan	62	Scotland	12
Poland	60	Holand-Netherlands	1
Columbia	59		
Taiwan	51		

Name: native_country, dtype: int64

https://en.wikipedia.org/wiki/List_of_countries_by_GNI_(nominal)_per_capita 에 따라 그룹으로 나눈다. (과적합방지)

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)

- native_country(국적)

```
income_01 = ['Jamaica',
 'Haiti',
 'Puerto-Rico',
 'Laos',
 'Thailand',
 'Ecuador',]

income_02 = ['Outlying-US(Guam-USVI-etc)',
 'Honduras',
 'Columbia',
 'Dominican-Republic',
 'Mexico',
 'Guatemala',
 'Portugal',
 'Trinadad&Tobago',
 'Nicaragua',
 'Peru',
 'Vietnam',
 'El-Salvador',]

income_03 = ['Poland',
 'Ireland',
 'South',
 'China',]

income_04 = [
    'United-States',
]
```

```
income_05 = [
 'Greece',
 'Scotland',
 'Cuba',
 'Hungary',
 'Hong',
 'Holand-Netherlands',
]
income_06 = [
 'Philippines',
 'Canada',
]
income_07 = [
 'England',
 'Germany',
]

income_08 = [
 'Italy',
 'India',
 'Japan',
 'France',
 'Yugoslavia',
 'Cambodia',
]

income_09 = [
 'Taiwan',
 'Iran',
]

income_other=['?', ]
```

잘 안보이시겠지만
복사+붙여넣기
하시면 됩니다!

```
income_01 = ['Jamaica',
 'Haiti',
 'Puerto-Rico',
 'Laos',
 'Thailand',
 'Ecuador',]

income_02 = ['Outlying-US(Guam-USVI-etc)',
 'Honduras',
 'Columbia',
 'Dominican-Republic',
 'Mexico',
 'Guatemala',
 'Portugal',
 'Trinadad&Tobago',
 'Nicaragua',
 'Peru',
 'Vietnam',
 'El-Salvador',]

income_03 = ['Poland',
 'Ireland',
 'South',
 'China',]

income_04 = [
    'United-States',
]
income_05 = [
 'Greece',
 'Scotland',
 'Cuba',
 'Hungary',
 'Hong',
 'Holand-Netherlands',
]
income_06 = [
 'Philippines',
 'Canada',
]
income_07 = [
 'England',
 'Germany',
]

income_08 = [
 'Italy',
 'India',
 'Japan',
 'France',
 'Yugoslavia',
 'Cambodia',
]

income_09 = [
 'Taiwan',
 'Iran',
]

income_other=['?', ]
```

4. 성인 인구 소득 예측 예제

- 칼럼별로 살펴보기 (object64형)
  - native_country(국적)

소득 구간별로 나누기 위한 함수 생성

```
def convert_country(x):
    if x in income_01:
        return 'income_01'
    elif x in income_02:
        return 'income_02'
    elif x in income_03:
        return 'income_03'
    elif x in income_04:
        return 'income_04'
    elif x in income_05:
        return 'income_05'
    elif x in income_06:
        return 'income_06'
    elif x in income_07:
        return 'income_07'
    elif x in income_08:
        return 'income_08'
    elif x in income_09:
        return 'income_09'
    else:
        return 'income_other'
```

```
all_data['country_bin'] = all_data['native_country'].apply(convert_country)
all_data['country_bin'].value_counts()
```

```
income_04       29170
income_02        1157
income_other      583
income_06         319
income_01         303
income_08         299
income_03         239
income_07         227
income_05         170
income_09          94
Name: country_bin, dtype: int64
```

4. 성인 인구 소득 예측 예제

- 최종 칼럼 선택

```
features = [
#     'id',
    'age',
    'workclass',
#     'fnlwgt',
    'fnlwgt_log',
    'education',
    'marital_status',
    'occupation',
    'relationship',
    'race',
    'sex',
#     'capital_gain',
    'capital_gain_log',
#     'capital_loss',
    'capital_loss_log',
    'hours_per_week',
#     'native_country',
    'country_bin'
]
```

```
label = ['income']
```

```
# One-Hot Encoding
all_data_dummies = pd.get_dummies(all_data[features + label])
```

```
all_data_dummies.head()
```

	age	fnlwgt_log	capital_gain_log	capital_loss_log	hours_per_week	income	workclass_?	workclass_Federal-gov	workclass_Local-gov	workclass_Other	...
0	40	12.034922	0.0	0.0	60	1.0	0	0	0	0	...
1	17	11.529065	0.0	0.0	20	0.0	0	0	0	0	...
2	18	12.775240	0.0	0.0	16	0.0	0	0	0	0	...
3	21	11.926088	0.0	0.0	25	0.0	0	0	0	0	...
4	24	11.713701	0.0	0.0	20	0.0	0	0	0	0	...

5 rows × 66 columns

처음에 합쳤던 데이터셋을 다시 train, test 데이터로 나누기

```
train_features = all_data_dummies.drop('income', axis=1).iloc[:len(train)]
test_features = all_data_dummies.drop('income', axis=1).iloc[len(train):]
```

```
train_features.shape, test_features.shape
```

```
((26049, 65), (6512, 65))
```

레이블 지정

```
train_label = train[label]
```

4. 성인 인구 소득 예측 예제

- 분류모델 적용

1) Random forest

```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import classification_report, f1_score, confusion_matrix

x_train, x_test, y_train, y_test = train_test_split(train_features, train_label, stratify=train_label,
                                                    test_size=0.2, random_state=6)
```

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(x_train, y_train)
print("Test Accuracy: {}%".format(round(rf.score(x_test, y_test)*100, 2)))
```

```
Test Accuracy: 85.3%
```

```
pred_rf = rf.predict(x_test)
print("Random Forest Classifier report: #n#n", classification_report(y_test, pred_rf))
```

```
Random Forest Classifier report:

              precision    recall  f1-score   support

           0       0.88      0.93      0.91      3949
           1       0.74      0.61      0.67      1261

    accuracy                           0.85      5210
   macro avg       0.81      0.77      0.79      5210
weighted avg       0.85      0.85      0.85      5210
```

Classification_report:
각각의 클래스를 양성(positive) 클래스로 보았을 때의 정밀도, 재현율, F1점수를 구하고 그 평균값으로 전체 모형의 성능을 평가한다.

4. 성인 인구 소득 예측 예제

- 분류모델 적용

2) xgboost

```
from xgboost import XGBClassifier

xgb_wrapper = XGBClassifier(random_state=42)
evals = [(x_test,y_test)]

xgb_wrapper.fit(x_train,y_train,early_stopping_rounds=100,eval_set=evals,verbose=True)

print("Test Accuracy: {}%".format(round(xgb_wrapper.score(x_test, y_test)*100, 2)))
```

```
[0]     validation_0-logloss:0.54529
[1]     validation_0-logloss:0.46540
[2]     validation_0-logloss:0.41560
[3]     validation_0-logloss:0.37944
[4]     validation_0-logloss:0.35726
[5]     validation_0-logloss:0.34107
[6]     validation_0-logloss:0.33046
[7]     validation_0-logloss:0.32190
[8]     validation_0-logloss:0.31618
[9]     validation_0-logloss:0.31242
[10]    validation_0-logloss:0.30907
[11]    validation_0-logloss:0.30684
[12]    validation_0-logloss:0.30534
[13]    validation_0-logloss:0.30403
[14]    validation_0-logloss:0.30236
[15]    validation_0-logloss:0.30015
[16]    validation_0-logloss:0.29927
[17]    validation_0-logloss:0.29820
Test Accuracy: 86.64%
```

```
pred_xgb = xgb_wrapper.predict(x_test)
print("XGBClassifier report: #n#n", classification_report(y_test, pred_xgb))
```

```
XGBClassifier report:

              precision    recall  f1-score   support

           0       0.89      0.94      0.91      3949
           1       0.78      0.63      0.69      1261

    accuracy                           0.87      5210
   macro avg       0.83      0.78      0.80      5210
weighted avg       0.86      0.87      0.86      5210
```

4. 성인 인구 소득 예측 예제

- 분류모델 적용

3) lgbm

```
from lightgbm import LGBMClassifier

lgbm_wrapper = LGBMClassifier(random_state=42)

evals = [(x_test,y_test)]

lgbm_wrapper.fit(x_train,y_train,early_stopping_rounds=100,eval_set=evals,verbose=True)
print("Test Accuracy: {}%".format(round(lgbm_wrapper.score(x_test, y_test)*100, 2)))
```

```
[1]     valid_0's binary_logloss: 0.51121
Training until validation scores don't improve for 100 rounds
[2]     valid_0's binary_logloss: 0.479711
[3]     valid_0's binary_logloss: 0.45501
[4]     valid_0's binary_logloss: 0.434664
[5]     valid_0's binary_logloss: 0.417514
[6]     valid_0's binary_logloss: 0.403037
[7]     valid_0's binary_logloss: 0.390581
[8]     valid_0's binary_logloss: 0.379714
[9]     valid_0's binary_logloss: 0.369946
[10]    valid_0's binary_logloss: 0.36195
[11]    valid_0's binary_logloss: 0.354916
[12]    valid_0's binary_logloss: 0.34849
[13]    valid_0's binary_logloss: 0.343188
[14]    valid_0's binary_logloss: 0.337497
[15]    valid_0's binary_logloss: 0.332999
Test Accuracy: 86.81%
```

```
preds = lgbm_wrapper.predict(x_test)
print("lgbm report: ₩n₩n", classification_report(y_test, preds))
```

```
lgbm report:

              precision    recall  f1-score   support

           0       0.89      0.94      0.92      3949
           1       0.78      0.64      0.70      1261

    accuracy                           0.87      5210
   macro avg       0.83      0.79      0.81      5210
weighted avg       0.86      0.87      0.86      5210
```

4. 성인 인구 소득 예측 예제

AutoML 사용해보기

```
from pycaret.classification import *
```

```
all_data_caret = all_data[features + label]
all_data_caret.head(1)
```

AutoML을 사용하기 위해, 원핫인코딩 되지 않은 데이터를 다시 불러옵니다.

	age	workclass	fnlwgt_log	education	marital_status	occupation	relationship	race	sex	capital_gain_log	cap
0	40	Private	12.034922	level_3	Married-civ-spouse	Sales	Husband	White	Male	0.0	

```
train_clean = all_data_caret[:len(train)]
test_clean = all_data_caret[len(train):]
```

```
setup(data = train_clean, target = 'income',session_id=42)
```

Processing:

Initiated	11:34:16
Status	Preprocessing Data

Following data types have been inferred automatically, if they are correct press enter to continue or type 'quit' otherwise.

4. 성인 인구 소득 예측 예제

AutoML 사용해보기

	Data Type
age	Numeric
workclass	Categorical
fnlwgt_log	Numeric
education	Categorical
marital_status	Categorical
occupation	Categorical
relationship	Categorical
race	Categorical
sex	Categorical
capital_gain_log	Numeric
capital_loss_log	Numeric
hours_per_week	Numeric
country_bin	Categorical
income	Label

데이터 타입을 확인하고,

엔터키를 눌러 넘어갑니다

	Description	Value
0	session_id	42
1	Target	income
2	Target Type	Binary
3	Label Encoded	0.0: 0, 1.0: 1
4	Original Data	(26049, 14)
5	Missing Values	False
6	Numeric Features	5
7	Categorical Features	8
8	Ordinal Features	False
9	High Cardinality Features	False
10	High Cardinality Method	None
11	Transformed Train Set	(18234, 63)
12	Transformed Test Set	(7815, 63)
13	Shuffle Train-Test	True
14	Stratify Train-Test	False
15	Fold Generator	StratifiedKFold
16	Fold Number	10
17	CPU Jobs	-1

4. 성인 인구 소득 예측 예제

AutoML 사용해보기

랜덤 포레스트

```
# 약 1~2분 정도 소요됩니다.
rf_automl = create_model('rf',fold=5) # 모델 생성
tuned_rf_automl = tune_model(rf_automl, optimize='F1',fold=5) # 튜닝
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.8552	0.9110	0.5226	0.8151	0.6369	0.5518	0.5731
1	0.8541	0.9101	0.5226	0.8094	0.6351	0.5492	0.5697
2	0.8563	0.9089	0.5260	0.8175	0.6401	0.5556	0.5767
3	0.8670	0.9161	0.5716	0.8284	0.6765	0.5963	0.6125
4	0.8560	0.9115	0.5158	0.8264	0.6352	0.5514	0.5751
Mean	0.8577	0.9115	0.5317	0.8194	0.6447	0.5608	0.5814
SD	0.0047	0.0025	0.0202	0.0071	0.0160	0.0178	0.0157

앞서 직접 만든 모델과 비교

```
rf_scores = cross_val_score(rf, train_features, train_label, scoring='f1', cv = 5)
print('평균 검증 f1 score:' ,np.round(np.mean(rf_scores),4))
```

```
평균 검증 f1 score: 0.6785
```

4. 성인 인구 소득 예측 예제

AutoML 사용해보기

XGBOOST

```
# 약 3분 정도 소요됩니다.
xgboost_automl = create_model('xgboost',fold=5) # 모델 생성
tuned_xgboost_automl = tune_model(xgboost_automl, optimize='F1',fold=5) # 튜닝
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.7719	0.9187	0.9120	0.5173	0.6601	0.5074	0.5537
1	0.7661	0.9184	0.9334	0.5102	0.6598	0.5039	0.5574
2	0.7732	0.9205	0.9233	0.5187	0.6642	0.5126	0.5613
3	0.7686	0.9257	0.9369	0.5133	0.6632	0.5089	0.5624
4	0.7836	0.9253	0.9379	0.5310	0.6781	0.5332	0.5823
Mean	0.7727	0.9217	0.9287	0.5181	0.6651	0.5132	0.5634
SD	0.0060	0.0032	0.0098	0.0071	0.0067	0.0104	0.0099

앞서 직접 만든 모델과 비교

```
xgb_wrapper_scores = cross_val_score(xgb_wrapper, train_features, train_label, scoring='f1', cv = 5)
print('평균 검증 f1 score:' ,np.round(np.mean(xgb_wrapper_scores),4))
```

```
평균 검증 f1 score: 0.7093
```

4. 성인 인구 소득 예측 예제

AutoML 사용해보기

LGBM

```
# 약 30초 정도 소요됩니다.
lgbm_automl = create_model('lightgbm',fold=5) #모델 생성
tuned_lgbm_automl = tune_model(lgbm_automl, optimize='F1',fold=5) # 튜닝
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.8656	0.9229	0.6354	0.7712	0.6968	0.6115	0.6162
1	0.8706	0.9243	0.6546	0.7775	0.7108	0.6282	0.6321
2	0.8706	0.9259	0.6354	0.7907	0.7046	0.6230	0.6291
3	0.8714	0.9288	0.6584	0.7787	0.7135	0.6313	0.6350
4	0.8714	0.9274	0.6467	0.7860	0.7096	0.6280	0.6329
Mean	0.8699	0.9259	0.6461	0.7808	0.7071	0.6244	0.6291
SD	0.0022	0.0021	0.0095	0.0068	0.0059	0.0070	0.0067

앞서 직접 만든 모델과 비교

```
lgbm_wrapper_scores = cross_val_score(lgbm_wrapper, train_features, train_label, scoring='f1', cv = 5)
print('평균 검증 f1 score:' ,np.round(np.mean(lgbm_wrapper_scores),4))
```

```
평균 검증 f1 score: 0.7119
```

4. 성인 인구 소득 예측 예제

AutoML 사용해보기
스태킹

```
# 약 2~3분 정도 소요됩니다.
best4models = compare_models(sort = 'F1', n_select = 4, fold=3)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
catboost	CatBoost Classifier	0.8687	0.9261	0.6434	0.7777	0.7042	0.6207	0.6253	20.6267
lightgbm	Light Gradient Boosting Machine	0.8677	0.9241	0.6421	0.7748	0.7022	0.6181	0.6226	0.5400
xgboost	Extreme Gradient Boosting	0.8650	0.9207	0.6423	0.7646	0.6981	0.6121	0.6159	3.0267
ada	Ada Boost Classifier	0.8600	0.9173	0.6247	0.7569	0.6844	0.5956	0.6001	0.7533
gbc	Gradient Boosting Classifier	0.8648	0.9214	0.5965	0.7959	0.6819	0.5982	0.6083	1.7667
rf	Random Forest Classifier	0.8527	0.9027	0.6177	0.7340	0.6708	0.5768	0.5804	1.8067
lr	Logistic Regression	0.8444	0.9011	0.5917	0.7183	0.6489	0.5501	0.5544	5.3167
lda	Linear Discriminant Analysis	0.8424	0.8959	0.5861	0.7141	0.6437	0.5438	0.5482	0.3300
et	Extra Trees Classifier	0.8333	0.8782	0.6093	0.6735	0.6398	0.5317	0.5328	2.0633
svm	SVM - Linear Kernel	0.7465	0.0000	0.8510	0.5175	0.6314	0.4665	0.5098	0.6000
ridge	Ridge Classifier	0.8418	0.0000	0.5383	0.7400	0.6232	0.5262	0.5369	0.2367
dt	Decision Tree Classifier	0.8072	0.7450	0.6240	0.5995	0.6114	0.4833	0.4836	0.3033
knn	K Neighbors Classifier	0.8109	0.8341	0.5712	0.6208	0.5949	0.4718	0.4726	3.9933
nb	Naive Bayes	0.6755	0.8606	0.9066	0.4222	0.5760	0.3656	0.4362	0.2700
qda	Quadratic Discriminant Analysis	0.6775	0.5089	0.1810	0.2649	0.2116	0.0202	0.0213	0.3000

4. 성인 인구 소득 예측 예제

AutoML 사용해보기
스태킹

```
# 약 2분 정도 소요됩니다.
stacker = stack_models(estimator_list = best4models[1:], meta_model = best4models[0], fold = 3)
```

	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.8644	0.9217	0.6175	0.7788	0.6888	0.6036	0.6103
1	0.8680	0.9268	0.6398	0.7778	0.7021	0.6184	0.6232
2	0.8720	0.9247	0.6662	0.7754	0.7167	0.6346	0.6377
Mean	0.8682	0.9244	0.6412	0.7773	0.7025	0.6189	0.6237
SD	0.0031	0.0021	0.0199	0.0014	0.0114	0.0127	0.0112

스태킹을 하면 성능이 높아질 가능성은 있으나
반드시 성능이 좋아지는 것은 아니다.

Q & A

감사합니다