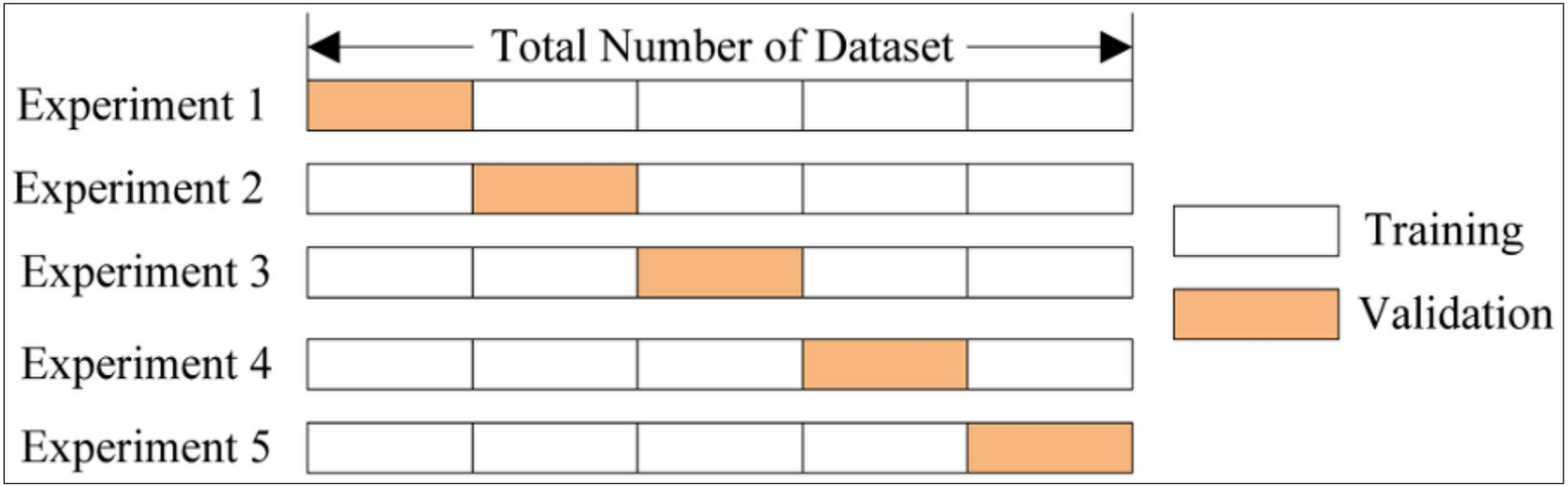
Explore Multi-Label Classification with an Enzyme Substrate Dataset __first place solution review

이다운, 유예송

K-fold


Total Number of Dataset
Experiment 1
Experiment 2
Experiment 3
Experiment 4
Experiment 5
Training
Validation

K-fold

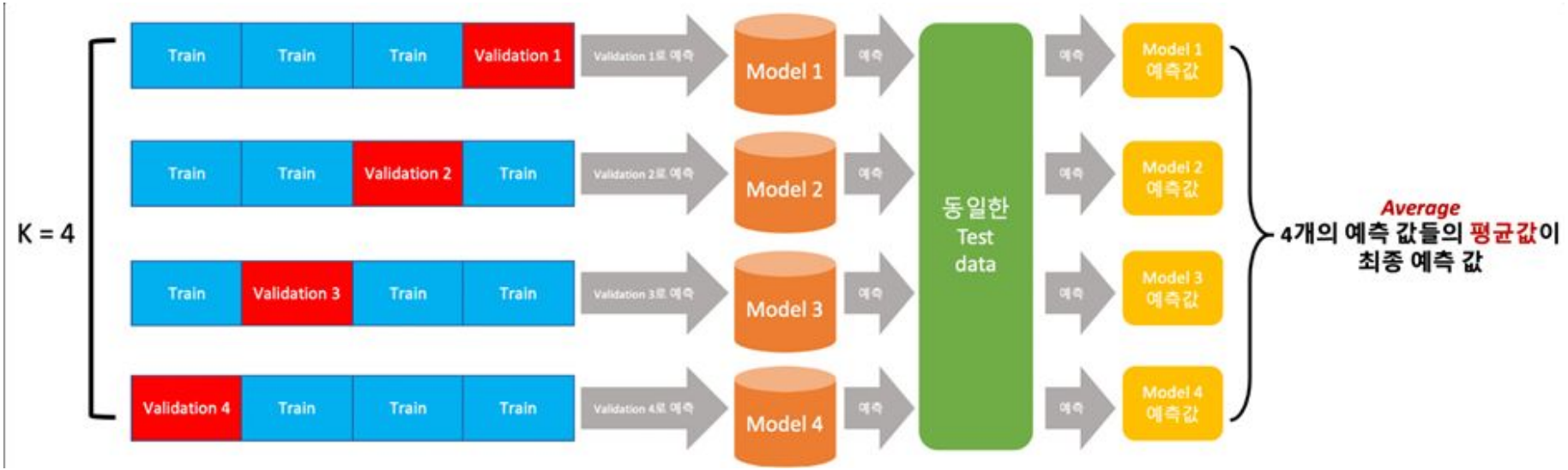
`RepeatedMultilabelStratifiedKFold` 사용→
다중 레이블(multi-label) 분류 문제를 고려한 계층적 분할을 수행하는 K-fold 교차 검증 방법

```
n_splits = 10
kf = RepeatedMultilabelStratifiedKFold(n_splits=n_splits, n_repeats=1, random_state=42)

for fn, (trn_idx, val_idx) in enumerate(kf.split(X, y)):
    print('Starting fold:', fn)
    # 'trn_idx'는 현재 fold에서의 학습용 인덱스
    # 'val_idx'는 현재 fold에서의 검증용 인덱스
    X_train, X_val = X.iloc[trn_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[trn_idx], y.iloc[val_idx]
```

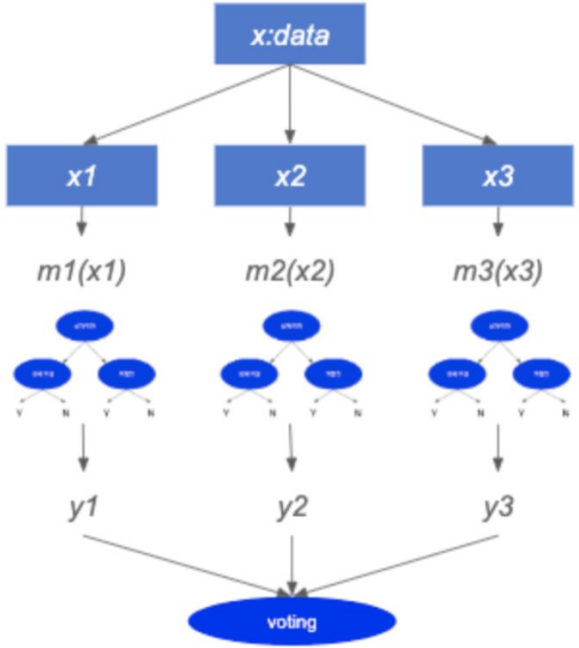
K-fold

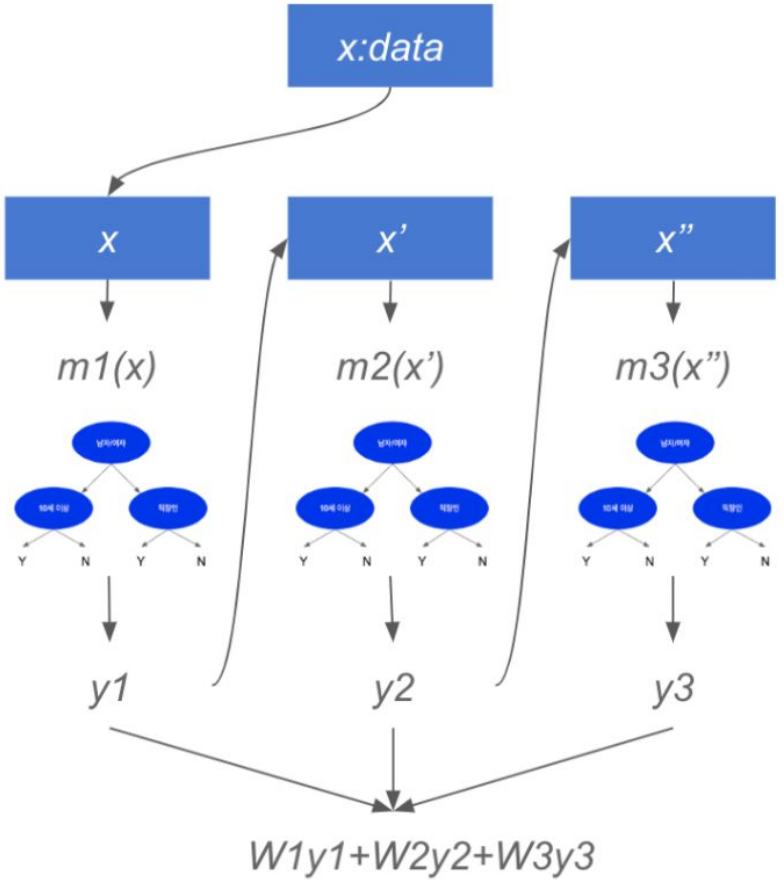
Out-of-Fold (OOF) 예측

OOF는 생성된 K개의 모델을 동일한 테스트 데이터에 적용시켜서 예측값을 내놓은 뒤 그 예측값을 평균내는 방법을 말한다.



XGBoost

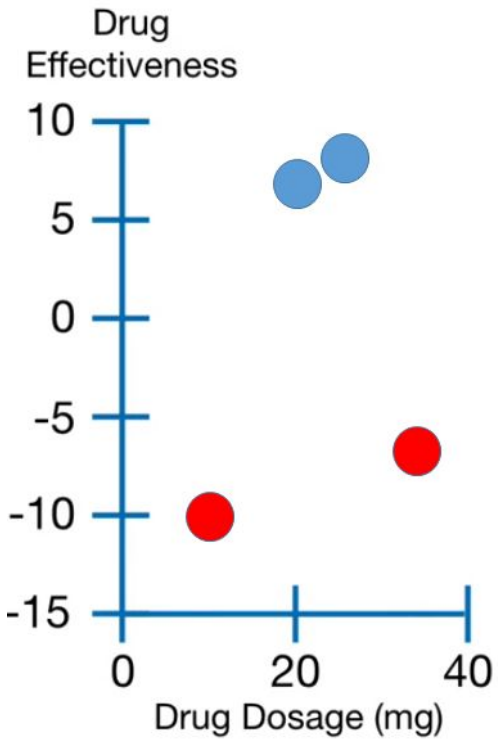

x:data
x1
x2
x3
m1(x1)
m2(x2)
m3(x3)
y1
y2
y3
voting

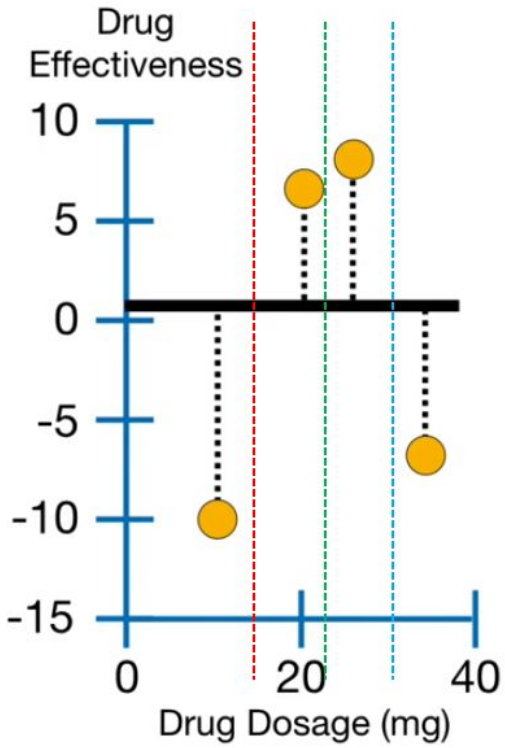

x:data
x
x'
x''
m1(x)
m2(x')
m3(x'')
y1
y2
y3
W1y1+W2y2+W3y3

XGBoost

1. XGBoost는 분류, 회귀 문제에 모두 사용할 수 있는 강력한 모델입니다.
2. 각 이터레이션에서 맞추지 못한 데이터에 **가중치를 부여**하여 모델을 학습시키는 부스팅(Boosting) 계열의 트리 모델입니다.
3. 강력한 **병렬 처리 성능**과 **자동 가지치기 알고리즘**이 적용되어 Gradient Boosting Model 대비 빠른 속도를 갖습니다.
4. **과적합 규제 기능(Regularization)의** 이점이 있습니다.
5. 또한 **자체 교차 검증 알고리즘**과 **결측치 처리 기능**을 가지고 있습니다.
6. **균형 트리 분할 방식**으로 모델을 학습하여 **대칭적인 트리를 형성**합니다.
7. Early Stopping 기능이 있습니다.

XGBoost

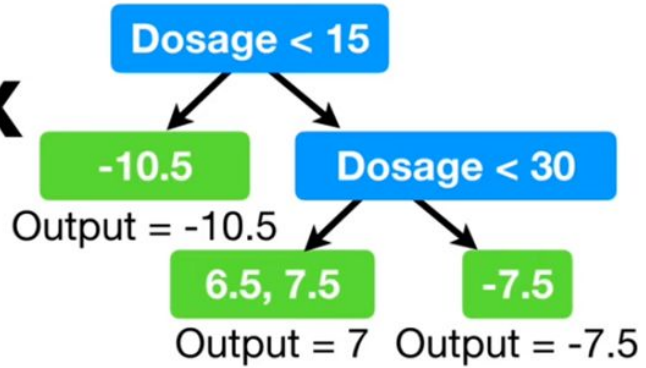

Drug
Effectiveness
10
5
0
-5
-10
-15
0
20
40
Drug Dosage (mg)

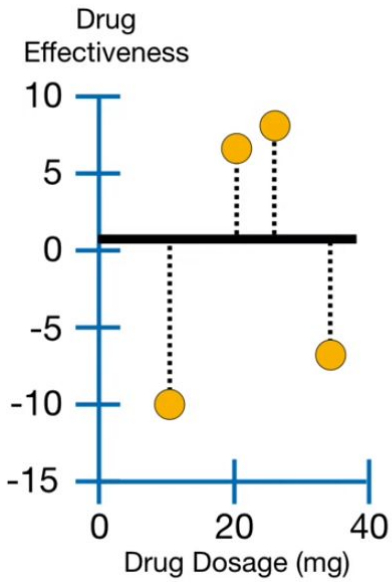

Drug
Effectiveness
10
5
0
-5
-10
-15
0
20
40
Drug Dosage (mg)

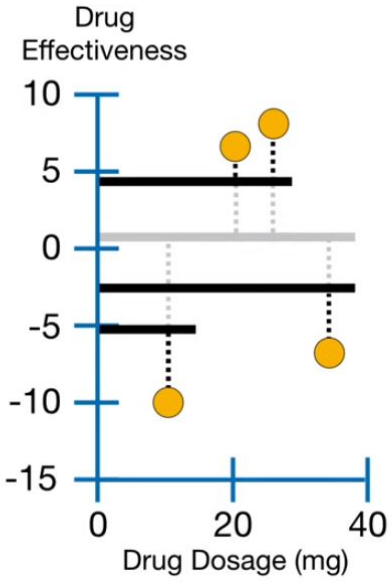
Predicted Drug Effectiveness

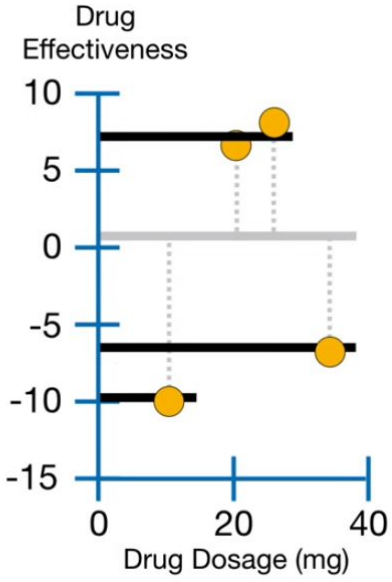
0.5

+ Learning Rate X









XGBoost

- **n_estimators (int)** : 내부에서 생성할 결정 트리의 개수
- **max_depth (int)** : 생성할 결정 트리의 높이
- learning_rate (float) : 훈련량, 학습 시 모델을 얼마나 업데이트할지 결정하는 값
- colsample_bytree (float) : 열 샘플링에 사용하는 비율
- subsample (float) : 행 샘플링에 사용하는 비율
- reg_alpha (float) : L1 정규화 계수
- reg_lambda (float) : L2 정규화 계수
- boosting_type (str) : 부스팅 방법 (gbdt / rf / dart / goss)
- random_state (int) : 내부적으로 사용되는 난수값
- n_jobs (int) : 병렬처리에 사용할 CPU 수

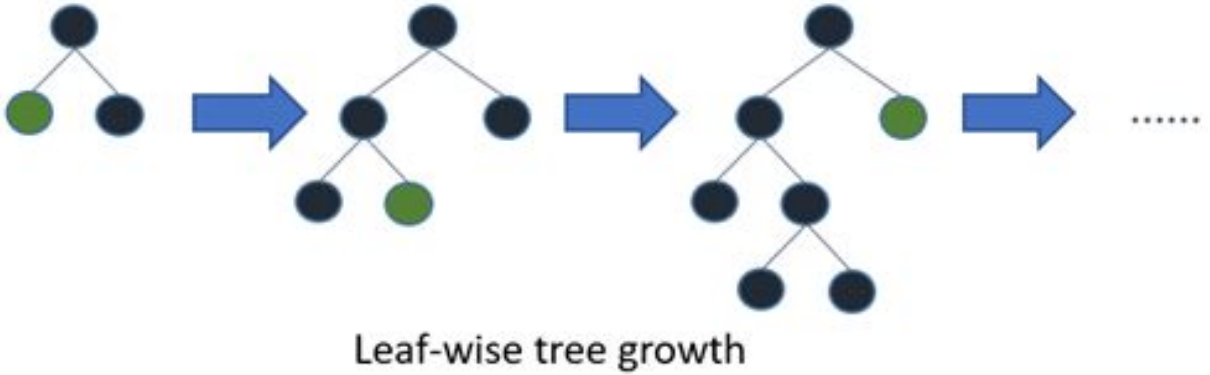
XGBoost

```
from xgboost import XGBClassifier

# XGBClassifier 모델 선언 후 Fitting
xgbc = XGBClassifier()
xgbc.fit(x_train, y_train)

# Fitting된 모델로 x_valid를 통해 예측을 진행
y_pred = xgbc.predict(x_valid)
```

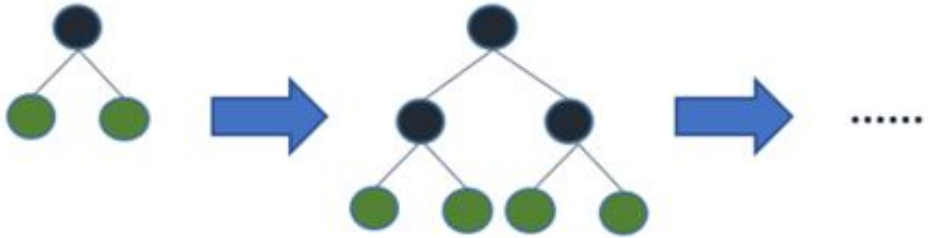
Light GBM

1. LightGBM은 분류, 회귀 문제에 모두 사용할 수 있는 강력한 모델입니다.
2. 각 이터레이션에서 맞추지 못한 데이터에 **가중치를 부여**하여 모델을 학습시키는 부스팅(Boosting) 계열의 트리 모델입니다.
3. 강력한 **병렬 처리 성능**과 **자동 가지치기 알고리즘**이 적용되어 Gradient Boosting Model 대비 빠른 속도를 갖습니다.
4. **과적합 규제 기능(Regularization)의** 이점이 있습니다.
5. 또한 **자체 교차 검증 알고리즘**과 **결측치 처리 기능**을 가지고 있습니다.
6. **리프 중심 트리 분할 방식**으로 **비대칭적인 트리**를 형성하여 모델을 학습하고, **예측 오류 손실을 최소화**합니다.
7. Early Stopping 기능이 있습니다.
8. 성능이 좋은 **XGBoost**와 성능은 비슷하지만 **속도가 훨씬 빠릅니다.**

Light GBM



Leaf-wise tree growth



Level-wise tree growth

Light GBM ← XGBoost의 파라미터와 동일!

- **n_estimators (int)** : 내부에서 생성할 결정 트리의 개수
- **max_depth (int)** : 생성할 결정 트리의 높이
- learning_rate (float) : 훈련량, 학습 시 모델을 얼마나 업데이트할지 결정하는 값
- colsample_bytree (float) : 열 샘플링에 사용하는 비율
- subsample (float) : 행 샘플링에 사용하는 비율
- reg_alpha (float) : L1 정규화 계수
- reg_lambda (float) : L2 정규화 계수
- boosting_type (str) : 부스팅 방법 (gbdt / rf / dart / goss)
- random_state (int) : 내부적으로 사용되는 난수값
- n_jobs (int) : 병렬처리에 사용할 CPU 수

Light GBM

```
from lightgbm import LGBMClassifier

# LGBMClassifier 모델 선언 후 Fitting
lgbc = LGBMClassifier()
lgbc.fit(x_train, y_train)

# Fitting된 모델로 x_valid를 통해 예측을 진행
# 0 or 1로 예측을 할 때
y_pred = lgbc.predict(x_valid)

# 0 ~ 1 사이의 확률값으로 예측을 할 때
y_pred_proba = lgbc.predict_proba(x_valid)
```

코드 분석

1st Place Winning Solution

- XGBoost, LightGBM
- Repeated Multi-label Stratified K-Fold

Data Preprocessing

```
train.drop(columns=["id"],inplace=True)
test.drop(columns=["id"],inplace=True)
mixed_desc.drop(columns=["CIDs"],inplace=True)
col="EC1_EC2_EC3_EC4_EC5_EC6"

mixed_desc[col.split("_")]= mixed_desc[col].str.split('_', expand=True).astype(int)
mixed_desc.drop(col, axis=1, inplace=True)

original = mixed_desc[train.columns]

train = pd.concat([train,original]).reset_index(drop=True)
train.drop(columns=col.split("_")[2:],inplace=True)
```

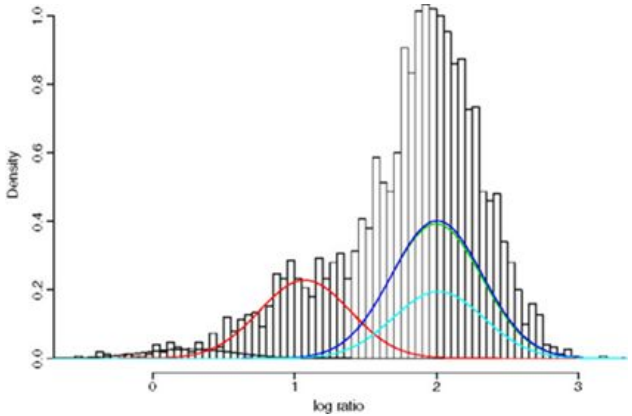
- 필요없는 컬럼 삭제 (ID, EC4~6)
- mixed_desc 데이터셋의 필요한 부분만 추출해 train과 병합

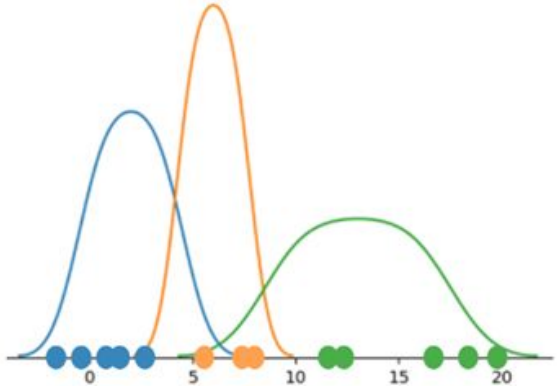
Data Preprocessing

```
from sklearn.mixture import GaussianMixture
def  get_gmm_class_feature(feat,n):
    gmm=GaussianMixture(n_components=n,random_state=42)
    gmm.fit(train[feat].values.reshape(-1,1))
    train[f'{feat}_class']=gmm.predict(train[feat].values.reshape(-1,1))
    test[f'{feat}_class']=gmm.predict(test[feat].values.reshape(-1,1))

get_gmm_class_feature("BertzCT",4)
get_gmm_class_feature("Chi1",4)
get_gmm_class_feature("Chi1n",3)
get_gmm_class_feature("Chi1v",3)
```

- Gaussian mixture for clustering and density estimation
- Gaussian Mixture을 이용하여 train과 test 데이터셋 군집화하여 새로운 컬럼으로 저장

GMM (Gaussian Mixture Model)





- 여러 가우시안 분포를 선형 결합하여 데이터셋을 모델링한다.
- 개별 데이터가 각 가우시안 분포에 속할 확률을 계산하여 가장 높은 확률을 가진 분포에 할당함으로써 데이터를 군집화할 수 있다.

Data Preprocessing

```
num=['BertzCT', 'Chi1', 'Chi1n', 'Chi1v', 'Chi2n', 'Chi2v', 'Chi3v', 'Chi4n',
       'EState_VSA1', 'EState_VSA2', 'ExactMolWt', 'FpDensityMorgan1',
       'FpDensityMorgan2', 'FpDensityMorgan3', 'HallKierAlpha',
       'HeavyAtomMolWt', 'Kappa3', 'MaxAbsEStateIndex', 'MinEStateIndex',
        'PEOE_VSA10', 'PEOE_VSA14', 'PEOE_VSA6', 'PEOE_VSA7',
       'PEOE_VSA8', 'SMR_VSA10', 'SMR_VSA5', 'SlogP_VSA3', 'VSA_EState9']

train['sum']=train[num].sum(axis=1)
train['mean']=train[num].mean(axis=1)
train['min']=train[num].min(axis=1)
train['max']=train[num].max(axis=1)
train['std']=train[num].std(axis=1)
train['var']=train[num].var(axis=1)
```

- 숫자형 변수에 대해 sum, mean, min/max, std, var 계산해 새로운 컬럼 생성

Feature Engineering

```
def fe(df):
    df['BertzCT_MaxAbsEStateIndex_Ratio']= df['BertzCT'] / (df['MaxAbsEStateIndex'] + 1e-12)
    df['BertzCT_ExactMolWt_Product']= df['BertzCT'] * df['ExactMolWt']
    df['NumHeteroatoms_FpDensityMorgan1_Ratio']= df['NumHeteroatoms'] / (df['FpDensityMorgan
1'] + 1e-12)
    df['VSA_EState9_EState_VSA1_Ratio']= df['VSA_EState9'] / (df['EState_VSA1'] + 1e-12)
    df['PEOE_VSA10_SMR_VSA5_Ratio']= df['PEOE_VSA10'] / (df['SMR_VSA5'] + 1e-12)
    df['Chi1v_ExactMolWt_Product']= df['Chi1v'] * df['ExactMolWt']
    df['Chi2v_ExactMolWt_Product']= df['Chi2v'] * df['ExactMolWt']
    df['Chi3v_ExactMolWt_Product']= df['Chi3v'] * df['ExactMolWt']
    df['EState_VSA1_NumHeteroatoms_Product']= df['EState_VSA1'] * df['NumHeteroatoms']
    df['PEOE_VSA10_Chi1_Ratio']= df['PEOE_VSA10'] / (df['Chi1'] + 1e-12)
```

- 변수 간의 관계와 비율을 새로운 컬럼에 저장

Generate Features

```
def generate_features(train, test, cat_cols, num_cols):
    df = pd.concat([train, test], axis = 0, copy = False)
    for c in cat_cols + num_cols:
        df[f'count_{c}'] = df.groupby(c)[c].transform('count')
    for c in cat_cols:
        for n in num_cols:
                df[f'mean_{n}_per_{c}'] = df.groupby(c)[n].transform('median')

    return df.iloc[:len(train),:], df.iloc[len(train):, :]
```

- 범주형과 숫자형 컬럼의 count를 저장한 변수 생성
- 범주형 변수의 각 클래스 별 중앙값 나타내는 mean 변수 생성

Model Training

```
# Define the classifiers
xgb_classifier = MultiOutputClassifier(XGBClassifier(**xgb_params))
lgbm_classifier = MultiOutputClassifier(LGBMClassifier(**lgbm_params))
#GBC_classifier = MultiOutputClassifier(GradientBoostingClassifier(n_estimators=100))

# Create the pipelines
xgb_clf = Pipeline([('classifier', xgb_classifier)])
lgbm_clf = Pipeline([('classifier', lgbm_classifier)])
#GBC_clf = Pipeline([('classifier', GBC_classifier)])
```

- 모델 정의 및 파이프라인 생성
- 파이프라인: 데이터 수집, 전처리, 모델 학습 등 머신러닝의 전체 과정을 순차적으로 처리하는 프로세스

Model Training

```
n_splits = 10
kf = RepeatedMultilabelStratifiedKFold(n_splits=n_splits, n_repeats=1, random_state=42)
train_losses_xgb = []
train_losses_lgbm = []
train_losses_GBC = []
```

- Repeated Multi-label Stratified K-fold

: 10개의 폴드로 나눠서 학습 진행함

Model Training

```
for fn, (trn_idx, val_idx) in enumerate(kf.split(X, y)):
    print('Starting fold:', fn)
    X_train, X_val = X.iloc[trn_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[trn_idx], y.iloc[val_idx]

    # Train and predict with XGBoost classifier
    xgb_clf.fit(X_train, y_train)
    train_preds_xgb = xgb_clf.predict_proba(X_train)
    train_preds_xgb = np.array(train_preds_xgb)[:, :, 1].T
    #train_loss_xgb = roc_auc_score(np.ravel(y_train), np.ravel(train_preds_xgb))
    #train_losses_xgb.append(train_loss_xgb)
```

- 각 폴드마다 train과 validation set으로 분할해 xgboost 모델 학습
- train_preds_xgb: 해당 폴드에서 train set 학습 후 예측 결과 저장

Model Training

```
val_preds_xgb = xgb_clf.predict_proba(X_val)
val_preds_xgb = np.array(val_preds_xgb)[:, :, 1].T
oof_preds_xgb[val_idx] = val_preds_xgb
loss_xgb = roc_auc_score(np.ravel(y_val), np.ravel(val_preds_xgb))
oof_losses_xgb.append(loss_xgb)
preds_xgb = xgb_clf.predict_proba(X_test)
preds_xgb = np.array(preds_xgb)[:, :, 1].T
test_preds_xgb += preds_xgb / n_splits
```

- 폴드의 validation set에 대해 예측 후 오차를 계산
- test_pred_xgb: 전체 테스트셋에 대한 예측값 저장

Model Training

```
lgbm_clf.fit(X_train, y_train)
train_preds_lgbm = lgbm_clf.predict_proba(X_train)
train_preds_lgbm = np.array(train_preds_lgbm)[:, :, 1].T
#train_loss_lgbm = roc_auc_score(np.ravel(y_train), np.ravel(train_preds_lgbm))
#train_losses_lgbm.append(train_loss_lgbm)

val_preds_lgbm = lgbm_clf.predict_proba(X_val)
val_preds_lgbm = np.array(val_preds_lgbm)[:, :, 1].T
oof_preds_lgbm[val_idx] = val_preds_lgbm

loss_lgbm = roc_auc_score(np.ravel(y_val), np.ravel(val_preds_lgbm))
oof_losses_lgbm.append(loss_lgbm)
preds_lgbm = lgbm_clf.predict_proba(X_test)
preds_lgbm = np.array(preds_lgbm)[:, :, 1].T
test_preds_lgbm += preds_lgbm / n_splits
```

- LGBM 모델에 대해 똑같은 과정 반복 후 test 예측값 저장

Model Training

```
overall_train_preds = (train_preds_xgb+train_preds_lgbm)/2
overall_train_loss = roc_auc_score(np.ravel(y_train), np.ravel(overall_train_preds))
overall_valid_preds = (val_preds_xgb+val_preds_lgbm)/2
overall_valid_loss = roc_auc_score(np.ravel(y_val), np.ravel(overall_valid_preds))
over_train.append(overall_train_loss)
over_valid.append(overall_valid_loss)
print("overall_train",overall_train_loss)
print("overall_valid",overall_valid_loss)

#GBC_clf = Pipeline([('classifier', GBC_classifier)])
```

- XGBoost와 LGBM 모델의 예측값을 합해 전체 loss 구함