



# 머신러닝 분반 1조 Modeling

발표자: 안수빈, 홍정기



# Table of Contents

## 1. Preprocessing

1-1. Variables transformation & NA

1-2. Feature selection and PCA

1-3. Oversampling

## 2. Modeling: GBM

## 3. Test score

# 1. Preprocessing

# 1-1. Variables transformation & NA

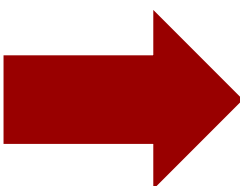
---

## 1) Variables transformation

- Class column 더미화
- Histogram 관련 변수들 기반으로 한 모멘트 파생변수 생성

# 1-1. Variables transformation & NA

```
df=pd.get_dummies(df, columns=["class"])
df=df.drop(['class_neg'],axis=1)
df.rename(columns = {"class_pos": "class"}, inplace=True)
```



class	class
neg	0
neg	0
neg	0
neg	0
neg	0
...	...
neg	0
neg	0
pos	1
neg	0
neg	0

- 원래의 데이터 셋에서는 class가 'neg', 'pos'로 되어 있음.
- pd.get\_dummies()를 이용하여 neg는 0의 값 pos는 1의 값을 갖는 더미변수 생성

# 1-1. Variables transformation & NA

ag_000	ag_001	ag_002	ag_003	ag_004	ag_005	ag_006	ag_007	ag_008	ag_009
0.0	0.0	0.0	0.0	51396.0	886464.0	1445974.0	463524.0	37460.0	288.0
0.0	0.0	0.0	0.0	452.0	42620.0	1139952.0	594268.0	42722.0	1356.0
0.0	0.0	0.0	10140.0	639334.0	9259336.0	7148984.0	676812.0	10432.0	114.0
0.0	0.0	0.0	0.0	112.0	66898.0	400152.0	66542.0	4032.0	0.0
0.0	0.0	0.0	0.0	35	0.0	77152.0	31582.0	0.0	0.0



ag_sum	ag_mean	ag_Var	ag_skewness
6.460162	1.844045	0.567685	0.203058
6.260398	2.351524	0.327623	0.638308
7.249080	1.443769	0.402210	0.238445
5.730569	2.013918	0.278783	0.312600
5.134840	2.026054	0.441509	-0.081959

- 분포의 모멘트(평균, 분산, 왜도)와 도수의 총합을 파생변수로 추가.
- 각 계급별 대푯값을 (-4, -3, ..., 4, 5)로 지정
- 분포의 성질을 잃지 않으며, 총 70개의 column을 28개의 column으로 축소가능.
- 총합은 log scale 적용.

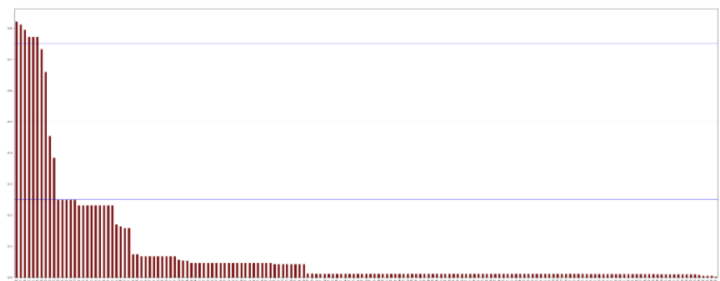
# 1-1. Variables transformation & NA

---

## 2) NA

- Missing Value EDA
- Missing Value Imputation (MICE & Iterative Imputer)

# 1-1. Variables transformation & NA



- 각 column의 결측치 비율을 barplot으로 시각화.

- 결측치 비율 0.5이상 columns drop

- 전체 122개 column중 99개의 column의 결측치 비율이 0.1보다 낮음.

→ 결측치 예측 필요.

	column	count	ratio
78	ak_000	46774	0.20596
79	ca_000	46774	0.20596
77	dm_000	3878	0.068035
1	df_000	3877	0.068018
112	dg_000	3877	0.068018
77	by_000	431	0.007561
77	ci_000	317	0.005561
77	cj_000	317	0.005561
73	ck_000	317	0.005561
73	bt_000	143	0.002509

	column	count	ratio
9	ak_000	4251	0.074579
46	ca_000	4213	0.073912
81	dm_000	3878	0.068035
74	df_000	3877	0.068018
75	dg_000	3877	0.068018
...	...	...	...
44	by_000	431	0.007561
54	ci_000	317	0.005561
55	cj_000	317	0.005561
56	ck_000	317	0.005561
40	bt_000	143	0.002509

99 rows × 3 columns



# 1-1. Variables transformation & NA

---

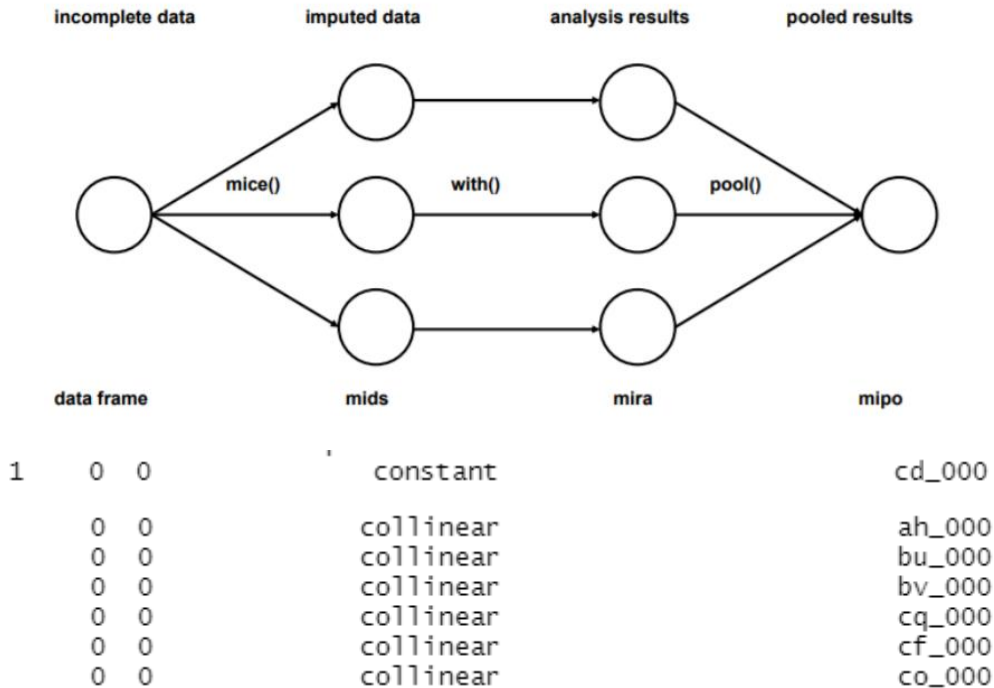
## Missing Value Imputation

- 평균 등의 값으로 대체하면 분산 등에 영향을 크게 줌.
- 따라서, 다중 대체법을 이용하여 결측치 예측.

# 1-1. Variables transformation & NA

## Multiple Imputation with Chained Equation

- 다른 모든 변수들을 이용하여 결측치 예측. ('CART' 모델을 적용)
- 결측치가 채워진 데이터셋을 여러 개 생성
- 만들어진 데이터셋을 하나로 통합
- 다른변수와 Collinearity를 갖거나 Constant인 경우 예측 불가



# 1-1. Variables transformation & NA

Multiple Imputation

incomplete data

imputed data

analysis results

pooled results

**Constant Column은 drop**  
**나머지 column에 대해 'Iterative Imputer'로**  
**결측치 예측 다시 시행**

Constant인 경우 예측 불가

$\bar{0}$   $\bar{0}$

collinear

co\_000

# 1-1. Variables transformation & NA

---

## Iterative Imputer

- Sklearn의 결측치 예측 모델
- 다른 모든 변수들을 바탕으로 결측치를 예측한다는 점이 MICE와 유사



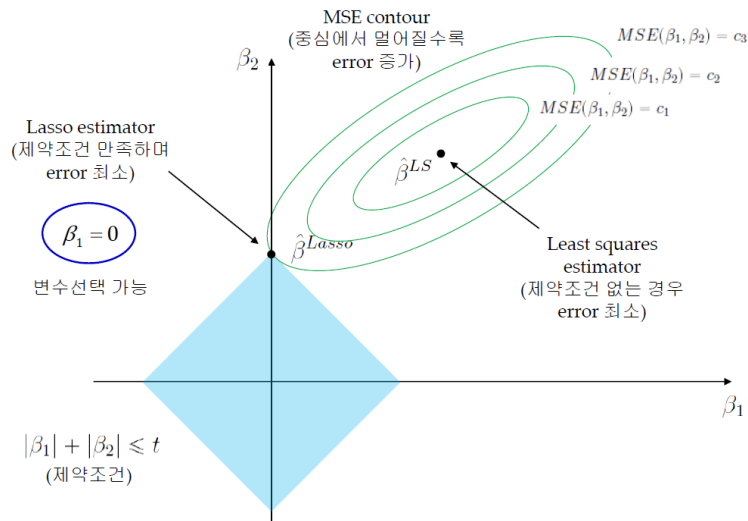
```
df.isnull().sum().sum()
```

모든 column에 대하여  
결측치 예측 완료 확인.

# 1-2. Feature selection and PCA

## 1) Feature selection

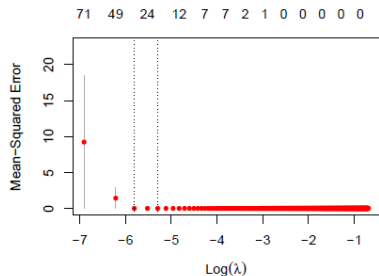
- Transformation 후, 열(변수) 제거 결과 남은 변수 119개
- LASSO regression 이용(R glmnet package)
- L1 규제를 통해 어떤 계수는 0으로 만듦으로써  
feature selection의 기능을 함



# 1-2. Feature selection and PCA

## Cross Validation for lambda

```
lambdas <- seq(0, 0.5, by=0.001)
cv_fit.lasso <- cv.glmnet(x, y, alpha=1, lambda=lambdas)
plot(cv_fit.lasso)
```



```
opt_lamb.1 <- .003
opt_lamb.1
```

```
## [1] 0.003
```

→ 0부터 0.5까지 0.001 간격으로 시퀀스를 형성하여 Mean-Squared Error를 최소화하는 최적의 lambda 값 도출 = 0.003

## Modeling

```
fin.lasso1 <- glmnet(x, y, alpha=1, lambda=opt_lamb.1)
coef(fin.lasso1)
```

→ Lambda값을 0.003으로 지정한 후, LASSO 모델링 진행

# 1-2. Feature selection and PCA

```
## 120 x 1 sparse Matrix of class "dgCMatrix"
##                               s0
## (Intercept) -1.063990e-02
## aa_000      .
## ac_000      .
## ad_000      .
## ae_000      .
## af_000      .
## ah_000      .
## ai_000      .
## aj_000      -6.236496e-09
## ak_000      .
## al_000      2.405215e-08
## am_0        2.309995e-09
## an_000      .
## ao_000      .
## ap_000      .
## aq_000      8.793495e-09
## ar_000      3.751133e-04
## as_000      .
## at_000      1.311539e-08
## au_000      1.393635e-08
## av_000      2.253389e-07
## ax_000      .
## bb_000      .
## bc_000      .
## bd_000      .
## be_000      .
## bf_000      .
## bg_000      .
## ...
```

이하 후락

→ 총 "31개" 의 변수가 선택됨(계수가 0이 아님)

```
## [1] "aj_000" "al_000" "am_0" "aq_000" "ar_000" "at_000" "au_000"
## [8] "av_000" "bi_000" "bj_000" "bs_000" "by_000" "cb_000" "cg_000"
## [15] "ci_000" "cj_000" "cm_000" "cz_000" "de_000" "dg_000" "di_000"
## [22] "do_000" "du_000" "dx_000" "dy_000" "ay_Var" "ba_mean" "cn_mean"
## [29] "cn_Var" "cs_sum" "ee_mean"
```

→ 31개 변수에 대한 상관계수 행렬에서 상관계수가 0.7이상으로 높게 나타나는 10개의 순서쌍 발견

→ Principal Component Analysis(PCA)를 통해 독립인 n개의 신규 변수를 만든다.

# 1-2. Feature selection and PCA

```
In [12]: > standardizing = StandardScaler()

In [13]: > feature_std = standardizing.fit_transform(train_feature)

In [14]: > cov_mat = np.dot(feature_std.T, feature_std)/(feature_std.shape[0]-1)

In [15]: > eigenvalue, eigenvector = np.linalg.eig(cov_mat)

In [16]: > eigenvalue.sort()
          > eigen_descending = eigenvalue[::-1]

In [17]: > eigen_descending[0:23]

Out[17]: array([7.22240509, 3.12755784, 1.80278543, 1.7140338 , 1.51582275,
                1.22935246, 1.16881451, 1.09551515, 1.00540805, 0.98680895,
                0.97643853, 0.93010927, 0.85592208, 0.82193403, 0.79413459,
                0.76256856, 0.71547539, 0.65857203, 0.51537809, 0.48377471,
                0.46918835, 0.40968167, 0.36108694])

In [18]: > eigenvalue, eigenvector = np.linalg.eig(cov_mat)

In [19]: > idx = []
          > for i in np.arange(23) :
              ind = np.where(eigenvalue == eigen_descending[i])[0][0]
              idx.append(ind)

In [20]: > princomp = np.dot(feature_std, eigenvector)

In [21]: > princomp_df = pd.DataFrame(princomp)

In [22]: > princomp_95 = princomp_df.loc[:, idx]
```

## 2) PCA

- 31개의 변수에 대해 PCA
- 데이터셋 전체 분산의 95% 이상 설명하는 23개의 주성분 선택
- Sklearn.decomposition의 PCA / R의 princomp()로도 가능



# 1-3. Oversampling

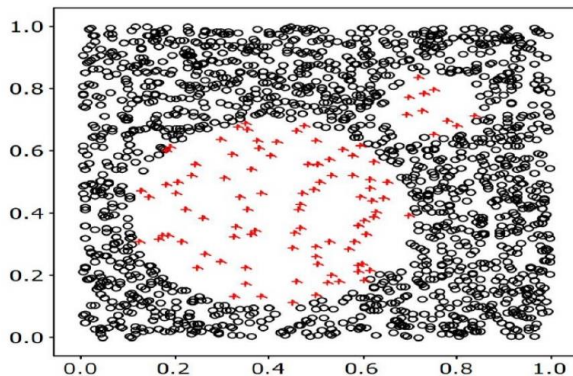
---

- SMOTE
- Borderline-SMOTE
- ADASYN

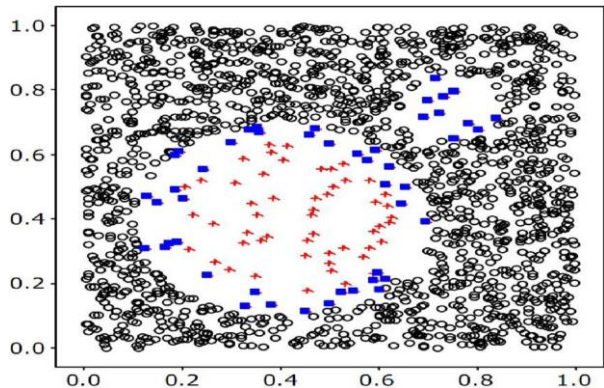
# 1-3-1. Borderline-SMOTE

- 모든 소수클래스를 대상으로 synthetic data를 만드는 SMOTE와 달리 경계값(boundary decision)에 있는 값만을 이용하여 SMOTE 오버샘플링하는 방식

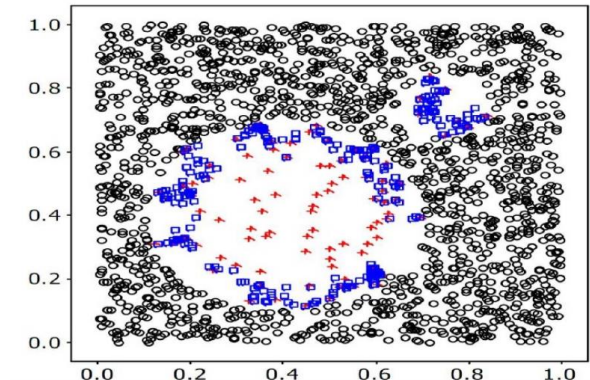
Circle data set (artificial)



Danger samples:



Borderline-Smote1 synthetic samples:



# 1-3-1. Borderline-SMOTE

---

- Borderline-SMOTE의 2가지 종류:

Borderline-SMOTE1/ Borderline-SMOTE2

1) Borderline-SMOTE1: 소수클래스의 데이터뿐만 아니라 decision boundary에서 misclassification을 일으키는 다수클래스의 데이터도 oversample.

2) Borderline-SMOTE2: 소수클래스에 속하는 값들만 가지고 oversample.

➡ Borderline-SMOTE는 misclassification이 경계선(boundary decision)에서 자주 일어날 때 효과적

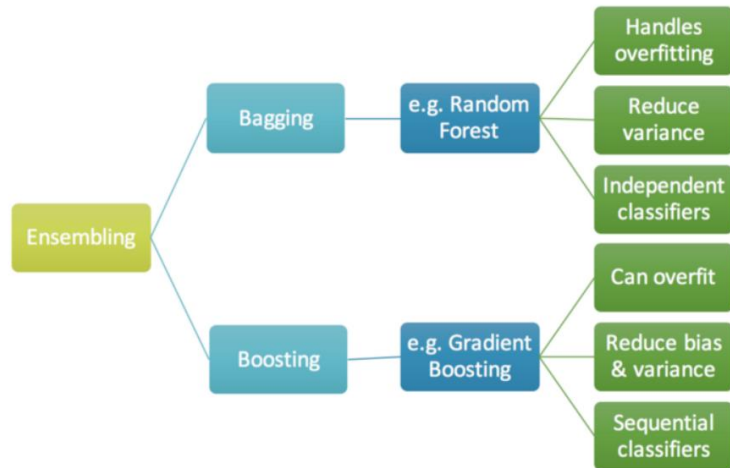
```
from imblearn.over_sampling import BorderlineSMOTE

oversample = BorderlineSMOTE(kind = 'borderline-1')
x2, y2 = oversample.fit_resample(x2, y2)
```

## 2. Modeling: GBM(Gradient Boosting Machine)

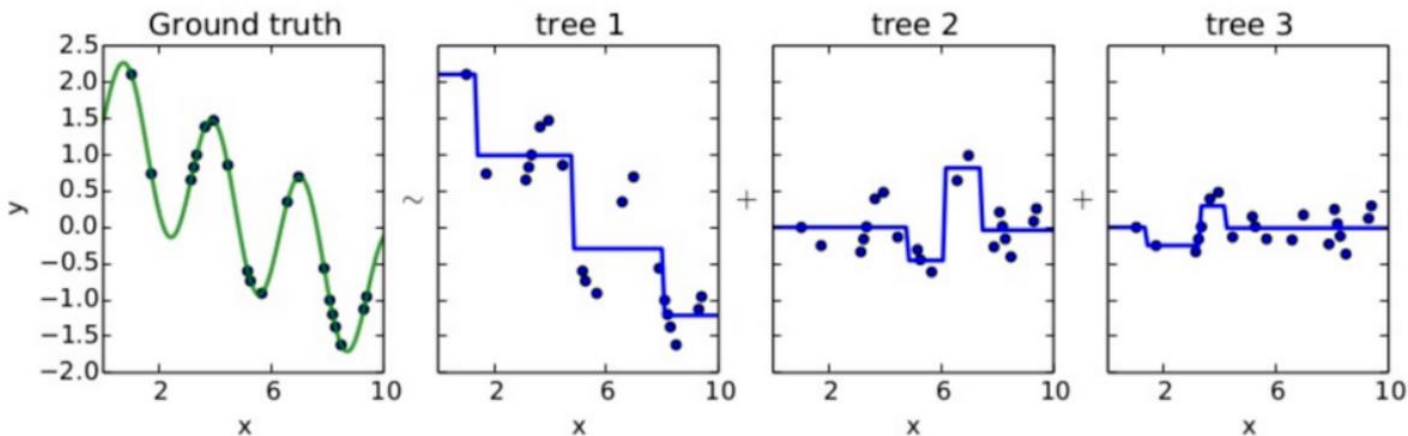
# GBM

- 회귀분석 또는 분류 분석을 수행할 수 있는 예측모형이며 예측모형의 앙상블 방법론 중 부스팅 계열에 속하는 알고리즘
- Gradient Boosting Algorithm을 구현한 패키지:  
LightGBM, CatBoost, XGBoost



# GBM

**Residual(negative gradient)에 fitting해서 다음 모델을 순차적으로 만들어 나가는 것**



# GBM

---

- Negative gradient(=pseudo-residual): 어떤 데이터 포인트에서 loss function이 줄어들기 위해  $f(x)$ 가 가려고하는 방향

→ 이 방향에 새로운 모델을 fitting해서 이것을 이전 모델과 결합하면  $f(x)$ 는 loss function이 줄어드는 방향으로 업데이트

**∴ Gradient boosting = gradient descent + boosting**

loss function을 줄이는 방향의 negative gradient를 얻고, 이를 활용해 boosting을 하는 것이기 때문에 gradient descent와 boosting이 결합된 방법

Input: training set  $\{(x_i, y_i)\}_{i=1}^n$ , a differentiable loss function  $L(y, F(x))$ , number of iterations  $M$ .

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, \gamma).$$

Baseline model

2. For  $m = 1$  to  $M$ :

1. Compute so-called *pseudo-residuals*.

$$r_{im} = - \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \quad \text{for } i = 1, \dots, n.$$

Negative gradient = pseudo-residual

2. Fit a base learner (e.g. tree)  $h_m(x)$  to pseudo-residuals, i.e. train it using the training set  $\{(x_i, r_{im})\}_{i=1}^n$ .

- 이 과정은  
성격도 비슷  
함
3. Compute multiplier  $\gamma_m$  by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)).$$

4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output  $F_M(x)$ .

대수적으로



# GBM

## `sklearn.ensemble.GradientBoostingClassifier`

주요 parameters:

- **learning\_rate**: 학습률
  - **n\_estimators**: boosting 단계의 개수(병렬적 tree의 개수)
  - **min\_samples\_split**: 하나의 node안에서 쪼개질 수 있는 최소한의 sample 수 (낮을 수록 overfit)
  - **min\_samples\_leaf**: terminal node 나 leaf에 남을 수 있는 최소한의 sample 수 (불균형 클래스를 다룰 때 낮은 값 설정)
  - **max\_depth**: tree의 깊이(높을 수록 overfit)
  - **max\_features**: tree를 나눌 때 고려하는 최대 개수의 변수 (높을 수록 overfit)
- Grid Search를 통해 각 parameter 튜닝 (expensive)
- Train/ Test AUC를 비교해 각 parameter 예측

# GBM

- **Baseline Model 측정 (default paramters)**

```
gbm = GradientBoostingClassifier(learning_rate=0.1, n_estimators=100, max_depth=3,  
                                min_samples_split=2, min_samples_leaf=1,  
                                subsample=1, max_features='sqrt', random_state=1)  
gbm.fit(X_train, y_train)  
predictors=list(X_train)  
print('Accuracy of the GBM on test set: {:.3f}'.format(gbm.score(X_val, y_val)))  
pred=gbm.predict(X_val)  
print(classification_report(y_val, pred))  
  
cm = confusion_matrix(y_val, pred).ravel()  
cm = pd.DataFrame(cm.reshape((1,4)), columns=['tn', 'fp', 'fn', 'tp'])  
display(cm)  
  
total_cost = 10*cm.fp + 500*cm.fn  
print(total_cost)
```

Accuracy of the GBM on test set: 0.978

	precision	recall	f1-score	support
0	0.98	0.97	0.98	11292
1	0.97	0.98	0.98	11082
accuracy			0.98	22374
macro avg	0.98	0.98	0.98	22374
weighted avg	0.98	0.98	0.98	22374

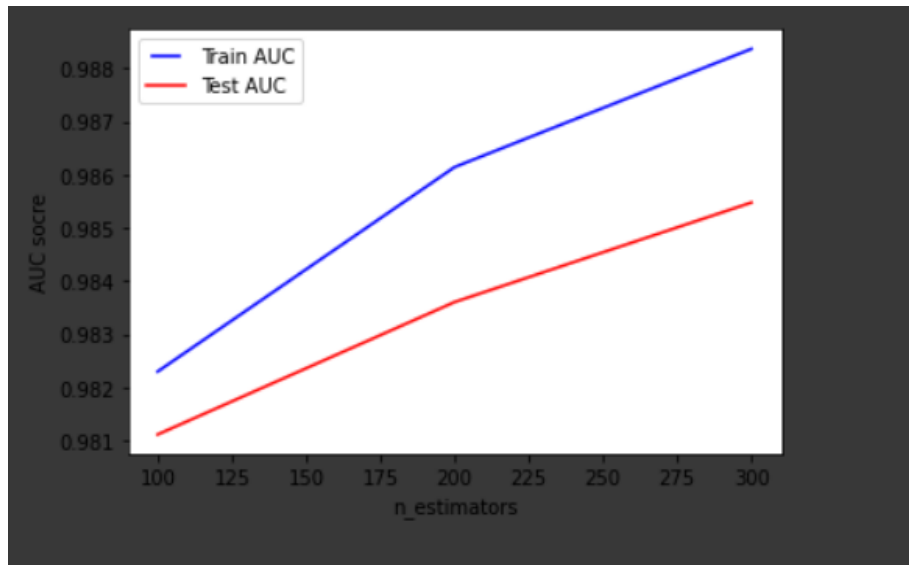
	tn	fp	fn	tp
0	10976	316	173	10909
1	89660			

dtype: int64

# GBM

- AUC를 이용해 각 **parameter** 비교 (**learning\_rate**, **n\_estimators**, **max\_depth**, **min\_samples\_split**, **min\_samples\_leaf**)

```
n_estimators = [100, 200, 300]
train_results = []
test_results = []
for eta in n_estimators:
    model = GradientBoostingClassifier(learning_rate=0.1, n_estimators=eta)
    model.fit(X_train, y_train)
    train_pred = model.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train, train_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = model.predict(X_val)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_val, y_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)
from matplotlib.legend_handler import HandlerLine2D
line1, = plt.plot(n_estimators, train_results, 'b', label='Train AUC')
line2, = plt.plot(n_estimators, test_results, 'r', label='Test AUC')
plt.legend(handler_map={line1: HandlerLine2D(numpoints=2)})
plt.ylabel('AUC score')
plt.xlabel('n_estimators')
plt.show()
```



# GBM

Grid Search 대신 SMOTE, Borderline-SMOTE, ADASYN 데이터에 대해 주요 4개 parameter의 train/test AUC를 비교해 가장 적합한 oversampling model과 parameter를 탐색한 결과:

```
Accuracy of the GBM on test set: 0.984
      precision    recall  f1-score   support

     0       0.99      0.98      0.98      11292
     1       0.98      0.99      0.98      11082

 accuracy          0.98      0.98      0.98      22374
 macro avg       0.98      0.98      0.98      22374
weighted avg       0.98      0.98      0.98      22374
```

```
      tn    fp    fn    tp
0 11043  249  106 10976
```

```
0      55490
dtype: int64
```

→ Borderline-Smote로 oversampling,  
learning\_rate=0.1, n\_estimators=100,  
max\_depth=4, min\_samples\_split=2,  
min\_samples\_leaf=5 로 train 한 모델 도출

→ Cost=55,490\$

# 3. Test Score

Accuracy Score: 0.97

Cost: 141410

Confusion matrix: 

18400	291
277	32

# Q&A

# 감사합니다