1. 문제 정의 및 계획: 한국인 영양조사 결과 데이터로부터 자살 고위험군 예측

```
import pandas as pd
import numpy as np
from collections import Counter
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
np.random.seed(3)
tf.random.set_seed(3)

df = pd.read_csv("/content/suicide_data.csv")
df
```

	ID	apt_t	sex	cfam	allownc	house	live_t	marri_2	tins	npins	...	N_CHOL	N_CHO	N_TDF	N_CA	N_FE	N
0	A739211515	1	2	3	20	2	3	1	10	1	...	220.911	213.937	12.957	469.710	9.405	2117
1	A739211516	1	1	3	20	2	3	1	10	1	...	352.617	274.917	20.682	372.172	14.404	3699
2	A739211517	1	2	3	20	2	3	3	20	1	...	136.681	272.175	19.292	266.870	13.276	2265
3	A739219614	1	1	4	20	1	3	1	20	1	...	41.882	243.170	25.090	358.145	20.760	4282
4	A739219615	1	2	4	20	1	3	1	20	1	...	140.545	288.721	30.959	487.907	14.351	2875
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
7375	P702300217	2	1	4	10	1	2	88	30	2	...	691.964	702.860	30.675	729.327	21.411	6821
7376	P702312314	2	2	2	20	1	2	3	10	2	...	227.482	297.520	42.728	503.535	19.751	3801
7377	P702322414	2	1	1	20	1	2	3	10	2	...	56.957	490.710	28.559	446.249	14.001	7703
7378	P702330414	2	1	1	10	1	2	2	30	2	...	2.523	335.034	14.751	183.639	11.132	1480
7379	P702336514	2	2	1	10	1	2	3	20	2	...	29.512	331.486	19.802	538.297	8.958	1840

7380 rows × 89 columns

2. 데이터 전처리

```
# 모름 또는 무응답으로 표기된 데이터를 결측값으로 대체
# 모름, 무응답으로 표기된 데이터는 8 또는 9(속성에 따라 88 또는 99)로 나와 있어 해당 데이터 또한 결측으로 표기
df = df.replace({'cfam':8, 'cfam':np.nan})
df = df.replace({'cfam':9, 'cfam':np.nan})
df = df.replace({'BD1_11':8, 'BD1_11':np.nan})
df = df.replace({'BD1_11':9, 'BD1_11':np.nan})
df = df.replace({'BE8_1':88, 'BE8_1':np.nan})
df = df.replace({'BE8_1':99, 'BE8_1':np.nan})
```

```
# 데이터의 타입을 숫자로 통일시키고, 숫자가 아닌 값들을 NaN으로 처리
print(Counter(df.dtypes))
df = df.iloc[:,1:].apply(pd.to_numeric, errors='coerce')
```

```
Counter({dtype('O'): 63, dtype('float64'): 16, dtype('int64'): 10})
```

```
# 데이터 타입이 숫자로 통일되었음을 확인
Counter(df.dtypes)
```

```
Counter({dtype('int64'): 10, dtype('float64'): 78})
```

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

```
apt_t        0
sex          0
cfam         0
allownc      0
house        0
            ...
N_NA       752
N_K        752
N_VITC     752
```

```
LF_SAFE    427
LF_S2      427
Length: 88, dtype: int64
```

```
# null 값이 있는 feature와 결측의 개수 출력하는 함수
def null_check(df):
    null = df.isnull().sum()

    null_col=[]

    for i in range(len(df.columns)):
        if (null[i]!=0):
            print(null.index[i],null[i])
            null_col.append(null.index[i])


# null 값 개수
null_check(df)
```

```
DM4_pr 476
D_8_2 476
D_8_4 476
DJ4_pr 476
DE1_pr 476
DE2_pr 476
DC1_pr 476
DC2_pr 476
DC3_pr 476
DC4_pr 476
DC5_pr 476
DC6_pr 476
DC7_pr 476
DF2_pr 476
DL1_pr 476
DJ8_pr 476
DH2_pr 476
DH3_pr 476
DN1_pr 476
DK8_pr 476
DK9_pr 476
DK4_pr 476
LQ4_00 476
LQ1_sb 476
LQ_1EQL 476
LQ_2EQL 476
LQ_3EQL 476
LQ_4EQL 476
LQ_5EQL 476
educ 476
EC1_1 476
BO1 476
BO1_1 476
BO2_1 476
BD1_11 476
incm 52
edu 968
occp 2025
HE_wt 409
HE_wc 413
HE_BMI 421
N_EN 752
N_WATER 752
N_PROT 752
N_FAT 752
N_SFA 752
N_MUFA 752
N_PUFA 752
N_CHOL 752
N_CHO 752
N_TDF 752
N_CA 752
N_FE 752
N_NA 752
N_K 752
N_VITC 752
LF_SAFE 427
LF_S2 427
```

```
# y_label에 해당하는 'mh_scicide' 변수 결측의 경우, 임의로 채울 수 없는 부분이므로 결측을 포함하는 행을 삭제
df1 = df.dropna(subset=['mh_suicide'])
null_check(df1)
```

```
mh_stress 1
L_OUT_FQ 643
LW_mt 2640
LW_oc 2640
HE_HP 845
```

```
HE_anem 411
O_chew_d 531
L_BR_FQ 643
L_LN_FQ 643
L_DN_FQ 643
DI4_pr 365
DM1_pr 365
incm 36
edu 398
occp 652
HE_wt 5
HE_wc 7
HE_BMI 9
N_EN 645
N_WATER 645
N_PROT 645
N_FAT 645
N_SFA 645
N_MUFA 645
N_PUFA 645
N_CHOL 645
N_CHO 645
N_TDF 645
N_CA 645
N_FE 645
N_NA 645
N_K 645
N_VITC 645
LF_SAFE 354
LF_S2 354
```

```
# 결측률이 10%를 훨씬 뛰어넘는 변수 LW_mt, LW_oc 삭제
df2 = df1.drop(['LW_mt','LW_oc'],axis='columns',inplace=False)
```

```
Counter(df2['mh_suicide'])
```

```
Counter({0.0: 5611, 1.0: 324})
```

```
# 숫자(수치형)는 평균으로 대체(numinputer) 문자(범주형)은 최빈값으로 대체(catimputer)
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler,StandardScaler,OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import FeatureUnion

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

```
# pipeline을 이용한 전처리
def pipeline(df, nums, cats):

    num_inputer = SimpleImputer(strategy='median')
    num_pipeline=Pipeline([
        ("select_numeric",DataFrameSelector(nums)),
        ("impute", num_inputer),
        ("scaler", StandardScaler())])

    cat_imputer = SimpleImputer(strategy='most_frequent')
    cat_pipeline = Pipeline([
        ("select_cat",DataFrameSelector(cats)),
        ("impute", cat_imputer)])
        #("encoder", OneHotEncoder())])

    preprocess_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline)])

    X=preprocess_pipeline.fit_transform(df)

    return X
```

```
nums = ['edu', 'occp', 'marri_1', 'LF_SAFE', 'LF_S2']
cats = ['allownc', 'house', 'LQ4_00', 'LQ1_sb', 'EC1_1', 'mh_stress', 'BE8_1']
```

```
X_data = pipeline(df2,nums,cats)
```

```
print(X_data[0])
```

```
[ 1.19595444  0.9107949  -0.55769414  0.65391279  0.22622831 20.
  2.          2.          2.          2.          0.          5.        ]
```

```
y_label = df2['mh_suicide']
```

```
print(X_data[0])
```



```
[ 1.19595444  0.9107949  -0.55769414  0.65391279  0.22622831 20.
  2.          2.          2.          2.          0.          5.        ]
```



```
y_label = df2['mh_suicide']
```



```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_data, y_label, test_size=0.3, random_state=0)
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)
X_test = np.asarray(X_test)
y_test = np.asarray(y_test)
```

```
# 훈련데이터와 테스트데이터의 분포와 label 확인
print('Train data shape: {0}'.format(X_train.shape))
print('Test data shape: {0}'.format(X_test.shape))
print('Train data label => %s' %Counter(y_train))
print('Test data label => %s' %Counter(y_test))
```

```
Train data shape: (4154, 12)
Test data shape: (1781, 12)
Train data label => Counter({0.0: 3926, 1.0: 228})
Test data label => Counter({0.0: 1685, 1.0: 96})
```

3. 기계학습 모델을 적용하여 데이터 분석

1) 의사결정트리

```
# 의사결정트리모델 적용
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 0.985
Accuracy on test set: 0.913
```

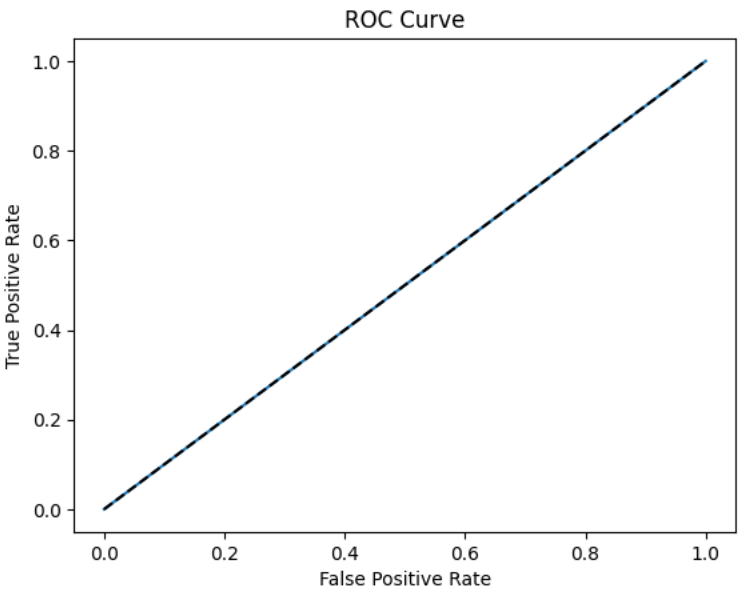
```
# pre-prunning max_depth를 3로 지정
tree1 = DecisionTreeClassifier(max_depth=3, random_state=0)
tree1.fit(X_train, y_train)
# 결과 - accuracy
print("Accuracy on training set: {:.3f}".format(tree1.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree1.score(X_test, y_test)))
```

```
Accuracy on training set: 0.945
Accuracy on test set: 0.946
```

```
# 결과 - recall, f1-score
from sklearn.metrics import classification_report
y_pred = tree1.predict(X_test)
print(classification_report(y_test, y_pred, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.95      1.00      0.97      1685
     class 1       0.00      0.00      0.00        96

    accuracy                           0.95      1781
   macro avg       0.47      0.50      0.49      1781
weighted avg       0.90      0.95      0.92      1781

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
```

```
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = tree1.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# 결과 - AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```




```
ROC AUC: 0.5
<Figure size 640x480 with 0 Axes>
```

가지치기를 2~5까지 적용해본 결과, max_depth=3일 때 test set에 대해 가장 높은 accuray를 보였다.

의사결정트리는 변수 중요도를 추출할 수 있다는 장점을 활용하여 변수의 중요도를 살펴보았다.

```
# 변수 중요도 추출
feature_importance = tree1.feature_importances_
print(feature_importance)
# occp, LF_S2, LQ4-00, mh-stress, BE8_1 이 중요 변수로 추출됨
```

```
[0.         0.09170477 0.         0.         0.28035967 0.
 0.         0.14808792 0.         0.         0.46298846 0.01685919]
```

2) 랜덤포레스트

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

```
Accuracy on training set: 0.985
Accuracy on test set: 0.938
```

```
y_pred0 = forest.predict(X_test)
print(classification_report(y_test, y_pred0, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.95      0.99      0.97      1685
     class 1       0.11      0.02      0.03        96

    accuracy                           0.94      1781
   macro avg       0.53      0.51      0.50      1781
```

```
weighted avg       0.90      0.94      0.92      1781
```

우리는 class1에 대한 높은 recall이 필요한데, 이 모델은 class1에 대해 낮은 재현율을 보였다. 따라서 랜덤 포레스트에서 사용할 트리의 수를 지정하는 n_estimators를 조정하여 재현율을 높여보았다.

```
forest1 = RandomForestClassifier(n_estimators=200, random_state=0)
forest1.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest1.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest1.score(X_test, y_test)))
```

```
Accuracy on training set: 0.985
Accuracy on test set: 0.937
```

```
y_pred1 = forest1.predict(X_test)
print(classification_report(y_test, y_pred1, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.95      0.99      0.97      1685
     class 1       0.14      0.03      0.05        96

    accuracy                           0.94      1781
   macro avg       0.54      0.51      0.51      1781
weighted avg       0.90      0.94      0.92      1781
```

n_estimators를 조정하여 사용하였는데도 그다지 성능이 좋아지지 않았다. 성능 향상을 위해 class weight를 사용해보았다.

```
class_weights = {0: 1, 1: 20}
forest2 = RandomForestClassifier(n_estimators=200, class_weight=class_weights, random_state=0)
forest2.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest2.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest2.score(X_test, y_test)))
```

```
Accuracy on training set: 0.970
Accuracy on test set: 0.921
```

```
y_pred2 = forest2.predict(X_test)
print(classification_report(y_test, y_pred2, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.95      0.97      0.96      1685
     class 1       0.13      0.08      0.10        96

    accuracy                           0.92      1781
   macro avg       0.54      0.53      0.53      1781
weighted avg       0.90      0.92      0.91      1781
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

forest = RandomForestClassifier(random_state=0)

# GridSeacrh
param_grid = {
    'n_estimators': [100, 200, 300],  # 나무의 개수
    'max_depth': [None, 10, 20, 30]  # 나무의 최대 깊이
}
grid_search = GridSearchCV(estimator=forest, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# 최적의 모델과 하이퍼파라미터를 출력
print("Best Parameters: ", grid_search.best_params_)

# 최적 모델의 정확도를 출력
best_rf = grid_search.best_estimator_
print("Accuracy on training set: {:.3f}".format(best_rf.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(best_rf.score(X_test, y_test)))
```

```
Best Parameters:  {'max_depth': 10, 'n_estimators': 100}
Accuracy on training set: 0.963
Accuracy on test set: 0.944
```

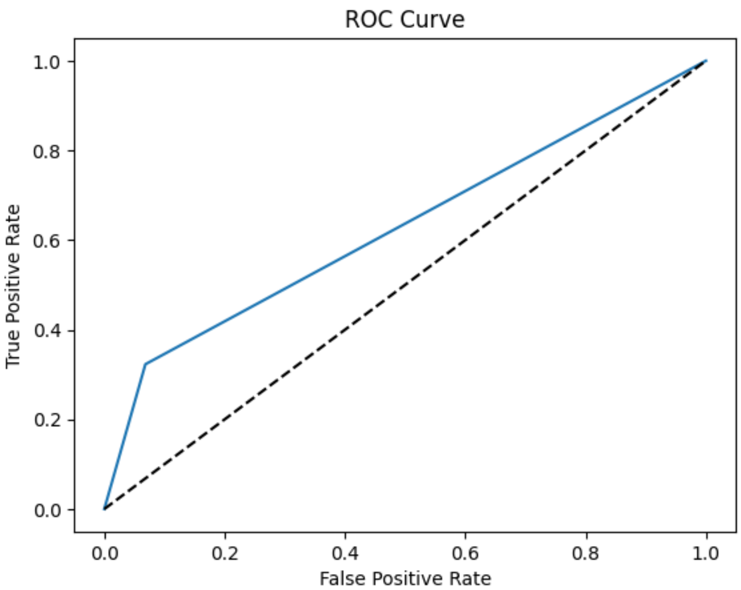
```
class_weights = {0: 1, 1: 20}
forest3 = RandomForestClassifier(n_estimators=100, max_depth=10, class_weight=class_weights, random_state=0)
forest3.fit(X_train, y_train)
# 결과 - accuracy
print("Accuracy on training set: {:.3f}".format(forest3.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest3.score(X_test, y_test)))
```

```
Accuracy on training set: 0.934
Accuracy on test set: 0.899
```

```
# 결과 - recall, f1 score
y_pred3 = forest3.predict(X_test)
print(classification_report(y_test, y_pred3, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.96      0.93      0.95      1685
     class 1       0.21      0.32      0.26        96

    accuracy                           0.90      1781
   macro avg       0.59      0.63      0.60      1781
weighted avg       0.92      0.90      0.91      1781
```

```
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = forest3.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# 결과 - AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```




```
ROC AUC: 0.6273337042532147
<Figure size 640x480 with 0 Axes>
```

3) XGBOOST

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

xgb_model = xgb.XGBClassifier()
xgb_model.fit(X_train, y_train)

y_pred = xgb_model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy on test set: {:.3f}".format(accuracy))
```

```
Accuracy on test set: 0.929
```

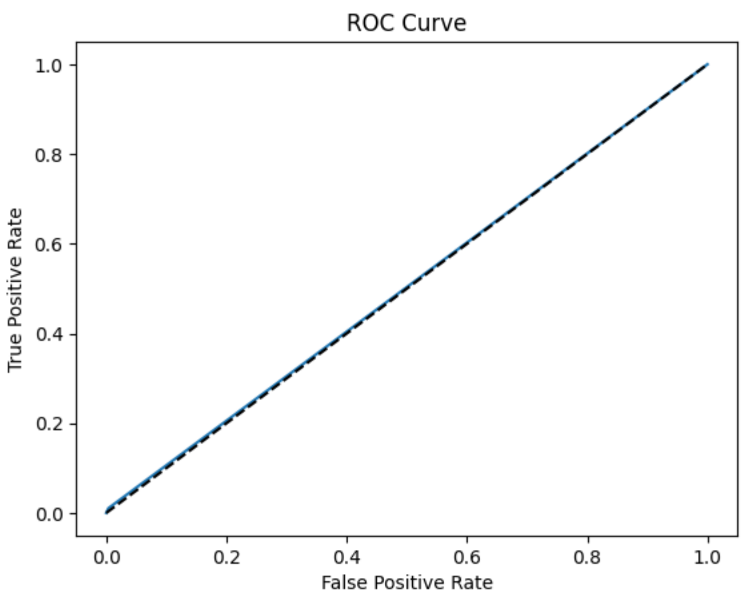
```
xgbest_model = xgb.XGBClassifier()

# 그리드 서치
param_grid = {
    'max_depth': [3, 4, 5],
    'learning_rate': [0.1, 0.01, 0.001],
    'n_estimators': [100, 200, 300],
}

grid_search = GridSearchCV(estimator=xgbest_model, param_grid=param_grid, scoring='accuracy', cv=3)
grid_search.fit(X_train, y_train)

# 최적의 하이퍼파라미터를 출력
print("Best Parameters: ", grid_search.best_params_)
# 최적 모델로 테스트 데이터로 예측
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
# 결과 - accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy on test set: {:.3f}".format(accuracy))
```

```
Best Parameters:  {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}
Accuracy on test set: 0.943
```

```
# 결과 - recall, f1 score
y_pred = best_model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.95      1.00      0.97      1685
     class 1       0.14      0.01      0.02        96

    accuracy                           0.94      1781
   macro avg       0.54      0.50      0.50      1781
weighted avg       0.90      0.94      0.92      1781
```

```
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = best_model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# 결과 - AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```




```
ROC AUC: 0.5034279179030663
<Figure size 640x480 with 0 Axes>
```

4) SVM

```
# 1. SVM, kernel = 'linear'로 선형분리 진행
import sklearn.svm as svm
import sklearn.metrics as mt
from sklearn.metrics import accuracy_score

# SVM은 선형과 비선형 모두 지원하므로 linear와 nonlinear 모두 가능
svm_clf =svm.SVC(kernel = 'linear')
```

SVM은 따로 train data와 test data를 분할하지 않고 진행하였다.

```
# 교차검증: 데이터가 적으므로 데이터를 여러 번 반복해서 나누고 여러 모델을 학습하여 성능을 평가하는 방법
import sklearn.metrics as mt
from sklearn.model_selection import cross_val_score, cross_validate

scores = cross_val_score(svm_clf, X_data, y_label, cv = 3)
print(scores)
# train data와 test data를 나누진 않지만 교차검증을 5번함
pd.DataFrame(cross_validate(svm_clf, X_data, y_label, cv = 3))

print('교차검증 평균: ', scores.mean())
```

```
[0.94542698 0.94539939 0.94539939]
교차검증 평균:  0.9454085899926988
```

```
# SVM, kernel = 'rbf'로 비선형분리 진행
svm_clf = svm.SVC(kernel = 'rbf')

# 교차검증
scores = cross_val_score(svm_clf, X_data, y_label, cv = 3)
print(scores)
pd.DataFrame(cross_validate(svm_clf, X_data, y_label, cv = 3))
print("교차검증 평균: ", scores.mean())
```

```
[0.94542698 0.94539939 0.94539939]
교차검증 평균:  0.9454085899926988
```

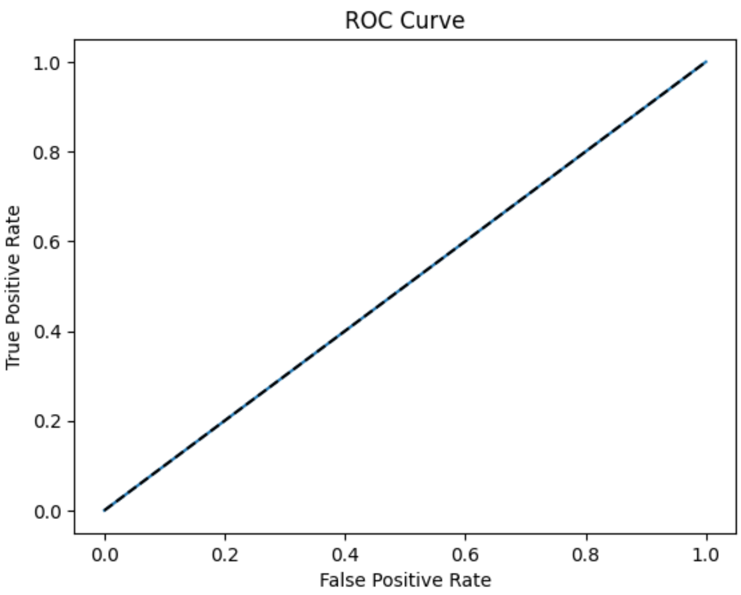
```
from sklearn.model_selection import GridSearchCV
# 하이퍼파라미터 그리드 설정
param_grid = {
    'C': [0.1, 1, 10],          # 여러 C 값
    'gamma': [0.1, 0.01, 0.001],  # 여러 gamma 값
    'kernel': ['rbf']              # RBF 커널 사용
}

# GridSearchCV 객체 생성
grid = GridSearchCV(svm.SVC(), param_grid, cv = 3)

# 그리드 서치 수행
grid.fit(X_data, y_label)

# 최적의 하이퍼파라미터와 그 때의 점수 출력
print("최적의 하이퍼파라미터:", grid.best_params_)
print("최적의 점수:", grid.best_score_)
```

```
최적의 하이퍼파라미터: {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
최적의 점수: 0.9454085899926988
```

```
# GridSearchCV 객체 생성
grid = GridSearchCV(svm.SVC(), param_grid, cv = 3)

# 그리드 서치 수행, 결과 - recall, f1 score
grid.fit(X_data, y_label)
y_pred = grid.predict(X_test)
print(classification_report(y_test, y_pred, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

     class 0       0.95      1.00      0.97      1685
     class 1       0.00      0.00      0.00        96

    accuracy                           0.95      1781
   macro avg       0.47      0.50      0.49      1781
weighted avg       0.90      0.95      0.92      1781

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = grid.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```




```
ROC AUC: 0.5
<Figure size 640x480 with 0 Axes>
```

5) DNN, 딥러닝

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 입력층 16개 은닉노드 + 은닉층1 8개 은닉노드
model = Sequential() # 딥러닝 생성
model.add(Dense(16, activation = 'relu', input_dim=12)) # Dense(은닉노드개수)으로 은닉층 추가
model.add(Dense(8, activation='relu')) # input_dim(입력노드개수)는 첫번째 정의할 때만 필요하고 여기선 안필요함
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer = 'adam',loss = 'binary_crossentropy', metrics=['accuracy'])

model.fit(X_train, y_train, epochs=50, batch_size=20)
```

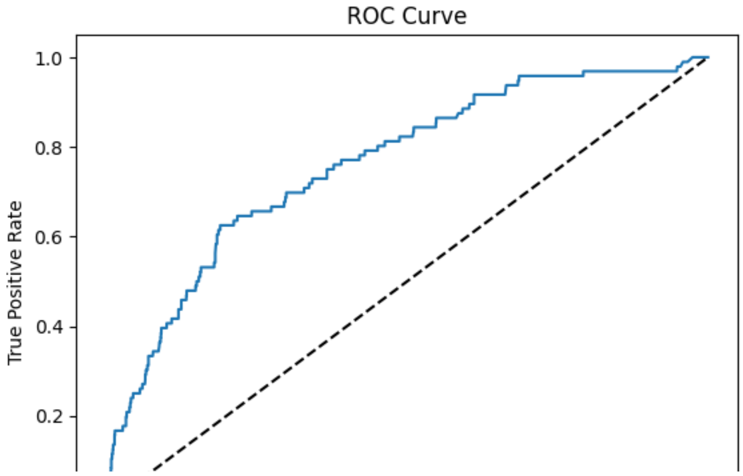
```
Epoch 34/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1722 - accuracy: 0.9454
Epoch 35/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1728 - accuracy: 0.9451
Epoch 36/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1722 - accuracy: 0.9463
Epoch 37/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1725 - accuracy: 0.9461
Epoch 38/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1719 - accuracy: 0.9468
Epoch 39/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1710 - accuracy: 0.9458
Epoch 40/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1719 - accuracy: 0.9468
Epoch 41/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1724 - accuracy: 0.9461
Epoch 42/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1735 - accuracy: 0.9466
Epoch 43/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1718 - accuracy: 0.9461
Epoch 44/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1712 - accuracy: 0.9463
Epoch 45/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1729 - accuracy: 0.9456
Epoch 46/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1695 - accuracy: 0.9470
Epoch 47/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1704 - accuracy: 0.9473
Epoch 48/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1699 - accuracy: 0.9463
Epoch 49/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1712 - accuracy: 0.9466
Epoch 50/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1723 - accuracy: 0.9466
<keras.src.callbacks.History at 0x7d72db996b60>
```

```
# 결과 - accuracy
score1 = model.evaluate(X_train, y_train)
score2 = model.evaluate(X_test, y_test)
print("Training Accuracy: %2f%%\n" % (score1[1]*100))
print("Test Accuracy: %2f%%\n" % (score2[1]*100))

# 결과 - recall, f1 score
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print(classification_report(y_test, y_pred))
```

```
130/130 [==============================] - 0s 2ms/step - loss: 0.1712 - accuracy: 0.9478
56/56 [==============================] - 0s 2ms/step - loss: 0.1878 - accuracy: 0.9467
Training Accuracy: 94.776118%

Test Accuracy: 94.665921%

56/56 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

         0.0       0.95      1.00      0.97      1685
         1.0       0.67      0.02      0.04        96

    accuracy                           0.95      1781
   macro avg       0.81      0.51      0.51      1781
weighted avg       0.93      0.95      0.92      1781
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
56/56 [==============================] - 0s 1ms/step
```




```
# 입력층 8개 은닉노드 + 은닉층1 16개 은닉노드
model1 = Sequential()
model1.add(Dense(8, activation = 'relu', input_dim=12))
model1.add(Dense(16, activation='relu'))
model1.add(Dense(1, activation='sigmoid'))

model1.compile(optimizer = 'adam',loss = 'binary_crossentropy', metrics=['accuracy'])

model1.fit(X_train, y_train, epochs=50, batch_size=20)
```

```
208/208 [==============================] - 1s 3ms/step - loss: 0.1760 - accuracy: 0.9446
Epoch 23/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1736 - accuracy: 0.9456
Epoch 24/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1747 - accuracy: 0.9466
Epoch 25/50
208/208 [==============================] - 0s 2ms/step - loss: 0.1750 - accuracy: 0.9468
Epoch 26/50
208/208 [==============================] - 1s 3ms/step - loss: 0.1751 - accuracy: 0.9451
Epoch 27/50
208/208 [==============================] - 1s 5ms/step - loss: 0.1740 - accuracy: 0.9456
Epoch 28/50
208/208 [==============================] - 1s 5ms/step - loss: 0.1735 - accuracy: 0.9454
```

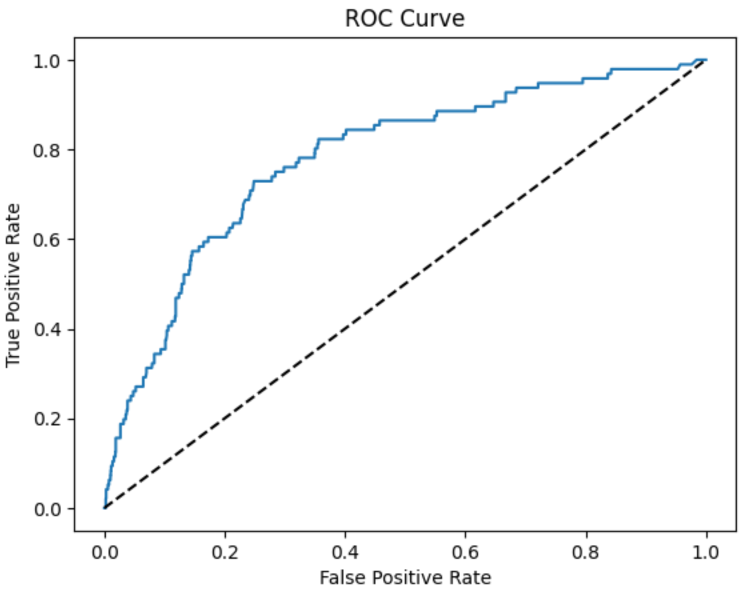
```
    208/208 [==============================] - 1s 3ms/step - loss: 0.1736 - accuracy: 0.9473
    <keras.src.callbacks.History at 0x7d72d8f7bdf0>
```

```
# 결과 - accuracy
score1 = model1.evaluate(X_train, y_train)
score2 = model1.evaluate(X_test, y_test)
print("Training Accuracy: %2f%%\n" % (score1[1]*100))
print("Test Accuracy: %2f%%\n" % (score2[1]*100))

# 결과 - recall, f1 score
y_pred_prob = model1.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print(classification_report(y_test, y_pred))
```

```
    130/130 [==============================] - 0s 2ms/step - loss: 0.1707 - accuracy: 0.9468
    56/56 [==============================] - 0s 2ms/step - loss: 0.1862 - accuracy: 0.9444
    Training Accuracy: 94.679826%

    Test Accuracy: 94.441324%

    56/56 [==============================] - 0s 2ms/step
                  precision    recall  f1-score   support

             0.0       0.95      1.00      0.97      1685
             1.0       0.29      0.02      0.04        96

        accuracy                           0.94      1781
       macro avg       0.62      0.51      0.51      1781
    weighted avg       0.91      0.94      0.92      1781
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = model1.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
    56/56 [==============================] - 0s 1ms/step
```




```
    ROC AUC: 0.7798126854599405
    <Figure size 640x480 with 0 Axes>
```

```
history = model1.fit(X_train, y_train, epochs=30, batch_size=64, validation_data=(X_test, y_test))
hist_df = pd.DataFrame(history.history)
hist_df

# y_vloss에 테스트셋의 오차를 저장합니다.
y_vloss = hist_df['val_loss']

# y_loss에 학습셋의 오차를 저장합니다.
y_loss = hist_df['loss']
```

```
# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

```
Epoch 1/30
65/65 [==============================] - 0s 4ms/step - loss: 0.1701 - accuracy: 0.9466 - val_loss: 0.1815 - val_accuracy: 0.9444
Epoch 2/30
65/65 [==============================] - 0s 3ms/step - loss: 0.1692 - accuracy: 0.9466 - val_loss: 0.1816 - val_accuracy: 0.9444
Epoch 3/30
65/65 [==============================] - 0s 3ms/step - loss: 0.1701 - accuracy: 0.9470 - val_loss: 0.1823 - val_accuracy: 0.9450
Epoch 4/30
65/65 [==============================] - 0s 3ms/step - loss: 0.1692 - accuracy: 0.9468 - val_loss: 0.1818 - val_accuracy: 0.9455
Epoch 5/30
65/65 [==============================] - 0s 3ms/step - loss: 0.1692 - accuracy: 0.9470 - val_loss: 0.1818 - val_accuracy: 0.9450
```

Optimization

```
Epoch 7/30
```

```
# L1 규제
from tensorflow.keras import regularizers, models
l1_model = models.Sequential()
l1_model.add(Dense(8, kernel_regularizer=regularizers.l1(0.01), activation='relu', input_dim=12))
l1_model.add(Dense(16, kernel_regularizer=regularizers.l1(0.01), activation='relu'))
l1_model.add(Dense(1, activation='sigmoid'))

l1_model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy',
                 metrics=['accuracy'])
```

```
Epoch 13/30
```

```
history = l1_model.fit(X_train, y_train, epochs=50, batch_size=64, validation_data=(X_test, y_test))
hist_df = pd.DataFrame(history.history)
hist_df

# y_vloss에 테스트셋의 오차를 저장합니다.
y_vloss = hist_df['val_loss']

# y_loss에 학습셋의 오차를 저장합니다.
y_loss = hist_df['loss']

# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

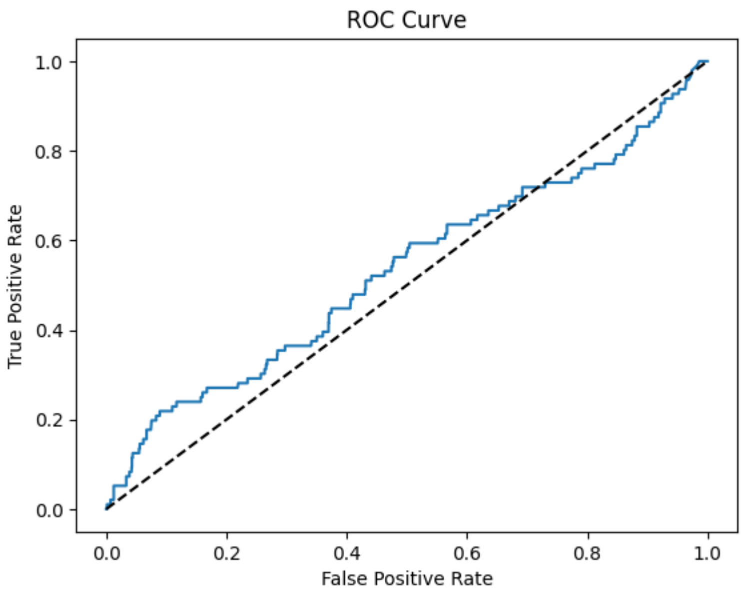
```
Epoch 1/50
65/65 [==============================] - 1s 5ms/step - loss: 0.8572 - accuracy: 0.9434 - val_loss: 0.7369 - val_accuracy: 0.9455
Epoch 2/50
65/65 [==============================] - 0s 3ms/step - loss: 0.6919 - accuracy: 0.9442 - val_loss: 0.6446 - val_accuracy: 0.9416
Epoch 3/50
65/65 [==============================] - 0s 3ms/step - loss: 0.6045 - accuracy: 0.9442 - val_loss: 0.5607 - val_accuracy: 0.9461
Epoch 4/50
65/65 [==============================] - 0s 3ms/step - loss: 0.5300 - accuracy: 0.9449 - val_loss: 0.4918 - val_accuracy: 0.9461
Epoch 5/50
65/65 [==============================] - 0s 3ms/step - loss: 0.4668 - accuracy: 0.9451 - val_loss: 0.4372 - val_accuracy: 0.9461
Epoch 6/50
65/65 [==============================] - 0s 3ms/step - loss: 0.4151 - accuracy: 0.9451 - val_loss: 0.3907 - val_accuracy: 0.9461
Epoch 7/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3773 - accuracy: 0.9451 - val_loss: 0.3601 - val_accuracy: 0.9461
Epoch 8/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3527 - accuracy: 0.9451 - val_loss: 0.3422 - val_accuracy: 0.9461
Epoch 9/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3351 - accuracy: 0.9451 - val_loss: 0.3263 - val_accuracy: 0.9461
Epoch 10/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3224 - accuracy: 0.9451 - val_loss: 0.3187 - val_accuracy: 0.9461
Epoch 11/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3122 - accuracy: 0.9451 - val_loss: 0.3064 - val_accuracy: 0.9461
Epoch 12/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3024 - accuracy: 0.9451 - val_loss: 0.3001 - val_accuracy: 0.9461
Epoch 13/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2977 - accuracy: 0.9451 - val_loss: 0.2959 - val_accuracy: 0.9461
Epoch 14/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2922 - accuracy: 0.9451 - val_loss: 0.2901 - val_accuracy: 0.9461
Epoch 15/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2862 - accuracy: 0.9451 - val_loss: 0.2854 - val_accuracy: 0.9461
Epoch 16/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2818 - accuracy: 0.9451 - val_loss: 0.2795 - val_accuracy: 0.9461
Epoch 17/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2780 - accuracy: 0.9451 - val_loss: 0.2758 - val_accuracy: 0.9461
Epoch 18/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2743 - accuracy: 0.9451 - val_loss: 0.2762 - val_accuracy: 0.9461
Epoch 19/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2717 - accuracy: 0.9451 - val_loss: 0.2717 - val_accuracy: 0.9461
Epoch 20/50
65/65 [==============================] - 1s 9ms/step - loss: 0.2691 - accuracy: 0.9451 - val_loss: 0.2688 - val_accuracy: 0.9461
Epoch 21/50
65/65 [==============================] - 1s 9ms/step - loss: 0.2663 - accuracy: 0.9451 - val_loss: 0.2663 - val_accuracy: 0.9461
Epoch 22/50
65/65 [==============================] - 0s 7ms/step - loss: 0.2644 - accuracy: 0.9451 - val_loss: 0.2626 - val_accuracy: 0.9461
Epoch 23/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2619 - accuracy: 0.9451 - val_loss: 0.2614 - val_accuracy: 0.9461
Epoch 24/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2596 - accuracy: 0.9451 - val_loss: 0.2618 - val_accuracy: 0.9461
Epoch 25/50
65/65 [==============================] - 1s 13ms/step - loss: 0.2583 - accuracy: 0.9451 - val_loss: 0.2567 - val_accuracy: 0.9461
Epoch 26/50
65/65 [==============================] - 1s 10ms/step - loss: 0.2556 - accuracy: 0.9451 - val_loss: 0.2541 - val_accuracy: 0.9461
Epoch 27/50
65/65 [==============================] - 1s 11ms/step - loss: 0.2537 - accuracy: 0.9451 - val_loss: 0.2526 - val_accuracy: 0.9461
Epoch 28/50
65/65 [==============================] - 1s 10ms/step - loss: 0.2523 - accuracy: 0.9451 - val_loss: 0.2505 - val_accuracy: 0.9461
Epoch 29/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2505 - accuracy: 0.9451 - val_loss: 0.2531 - val_accuracy: 0.9461
Epoch 30/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2497 - accuracy: 0.9451 - val_loss: 0.2497 - val_accuracy: 0.9461
Epoch 31/50
65/65 [==============================] - 0s 7ms/step - loss: 0.2472 - accuracy: 0.9451 - val_loss: 0.2472 - val_accuracy: 0.9461
Epoch 32/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2459 - accuracy: 0.9451 - val_loss: 0.2434 - val_accuracy: 0.9461
Epoch 33/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2432 - accuracy: 0.9451 - val_loss: 0.2444 - val_accuracy: 0.9461
Epoch 34/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2415 - accuracy: 0.9451 - val_loss: 0.2401 - val_accuracy: 0.9461
Epoch 35/50
65/65 [==============================] - 0s 5ms/step - loss: 0.2409 - accuracy: 0.9451 - val_loss: 0.2396 - val_accuracy: 0.9461
Epoch 36/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2403 - accuracy: 0.9451 - val_loss: 0.2385 - val_accuracy: 0.9461
Epoch 37/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2393 - accuracy: 0.9451 - val_loss: 0.2372 - val_accuracy: 0.9461
Epoch 38/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2379 - accuracy: 0.9451 - val_loss: 0.2366 - val_accuracy: 0.9461
Epoch 39/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2367 - accuracy: 0.9451 - val_loss: 0.2354 - val_accuracy: 0.9461
Epoch 40/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2359 - accuracy: 0.9451 - val_loss: 0.2343 - val_accuracy: 0.9461
Epoch 41/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2352 - accuracy: 0.9451 - val_loss: 0.2334 - val_accuracy: 0.9461
Epoch 42/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2344 - accuracy: 0.9451 - val_loss: 0.2328 - val_accuracy: 0.9461
Epoch 43/50
```

```
# 결과 - accuracy
score1 = l1_model.evaluate(X_train, y_train)
score2 = l1_model.evaluate(X_test, y_test)
print("Training Accuracy: %2f%%\n" % (score1[1]*100))
print("Test Accuracy: %2f%%\n" % (score2[1]*100))

# 결과 - recall, f1 score
y_pred_prob = l1_model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print(classification_report(y_test, y_pred))
```

```
130/130 [==============================] - 0s 2ms/step - loss: 0.2251 - accuracy: 0.9451
56/56 [==============================] - 0s 3ms/step - loss: 0.2246 - accuracy: 0.9461
Training Accuracy: 94.511312%

Test Accuracy: 94.609767%

56/56 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

         0.0       0.95      1.00      0.97      1685
         1.0       0.00      0.00      0.00        96

    accuracy                           0.95      1781
   macro avg       0.47      0.50      0.49      1781
weighted avg       0.90      0.95      0.92      1781

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = l1_model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
56/56 [==============================] - 0s 1ms/step
```




```
ROC AUC: 0.5327553165182988
<Figure size 640x480 with 0 Axes>
```

```
# L2 규제
l2_model = models.Sequential()
l2_model.add(Dense(8, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu', input_dim=12))
l2_model.add(Dense(16, kernel_regularizer=regularizers.l2(0.001),
```

```
                          activation='relu'))
l2_model.add(Dense(1, activation='sigmoid'))

l2_model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy',
                 metrics=['accuracy'])

# L1은 절대값, L2는 제곱만큼 보정을 해줌


history = l2_model.fit(X_train, y_train, epochs=50, batch_size=64, validation_data=(X_test, y_test))
hist_df = pd.DataFrame(history.history)
hist_df

# y_vloss에 테스트셋의 오차를 저장합니다.
y_vloss = hist_df['val_loss']

# y_loss에 학습셋의 오차를 저장합니다.
y_loss = hist_df['loss']

# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

```
Epoch 1/50
65/65 [==============================] - 1s 7ms/step - loss: 0.2636 - accuracy: 0.9429 - val_loss: 0.2304 - val_accuracy: 0.9461
Epoch 2/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2273 - accuracy: 0.9449 - val_loss: 0.2253 - val_accuracy: 0.9455
Epoch 3/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2222 - accuracy: 0.9449 - val_loss: 0.2213 - val_accuracy: 0.9455
Epoch 4/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2189 - accuracy: 0.9449 - val_loss: 0.2186 - val_accuracy: 0.9455
Epoch 5/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2159 - accuracy: 0.9449 - val_loss: 0.2188 - val_accuracy: 0.9455
Epoch 6/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2136 - accuracy: 0.9449 - val_loss: 0.2153 - val_accuracy: 0.9455
Epoch 7/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2114 - accuracy: 0.9449 - val_loss: 0.2154 - val_accuracy: 0.9455
Epoch 8/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2104 - accuracy: 0.9446 - val_loss: 0.2135 - val_accuracy: 0.9444
Epoch 9/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2078 - accuracy: 0.9446 - val_loss: 0.2106 - val_accuracy: 0.9444
Epoch 10/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2075 - accuracy: 0.9451 - val_loss: 0.2145 - val_accuracy: 0.9444
Epoch 11/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2068 - accuracy: 0.9454 - val_loss: 0.2085 - val_accuracy: 0.9444
Epoch 12/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2033 - accuracy: 0.9449 - val_loss: 0.2077 - val_accuracy: 0.9450
Epoch 13/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2031 - accuracy: 0.9456 - val_loss: 0.2088 - val_accuracy: 0.9444
Epoch 14/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2032 - accuracy: 0.9454 - val_loss: 0.2068 - val_accuracy: 0.9439
Epoch 15/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2015 - accuracy: 0.9451 - val_loss: 0.2060 - val_accuracy: 0.9439
Epoch 16/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2009 - accuracy: 0.9456 - val_loss: 0.2045 - val_accuracy: 0.9444
Epoch 17/50
65/65 [==============================] - 0s 3ms/step - loss: 0.2007 - accuracy: 0.9454 - val_loss: 0.2034 - val_accuracy: 0.9444
Epoch 18/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1994 - accuracy: 0.9454 - val_loss: 0.2083 - val_accuracy: 0.9444
Epoch 19/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1990 - accuracy: 0.9454 - val_loss: 0.2049 - val_accuracy: 0.9439
Epoch 20/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1984 - accuracy: 0.9454 - val_loss: 0.2037 - val_accuracy: 0.9439
Epoch 21/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1973 - accuracy: 0.9454 - val_loss: 0.2032 - val_accuracy: 0.9439
Epoch 22/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1971 - accuracy: 0.9456 - val_loss: 0.2005 - val_accuracy: 0.9450
Epoch 23/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1963 - accuracy: 0.9454 - val_loss: 0.2025 - val_accuracy: 0.9450
Epoch 24/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1954 - accuracy: 0.9456 - val_loss: 0.2045 - val_accuracy: 0.9444
Epoch 25/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1955 - accuracy: 0.9454 - val_loss: 0.2001 - val_accuracy: 0.9444
Epoch 26/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1943 - accuracy: 0.9458 - val_loss: 0.1980 - val_accuracy: 0.9444
Epoch 27/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1934 - accuracy: 0.9458 - val_loss: 0.1981 - val_accuracy: 0.9444
Epoch 28/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1933 - accuracy: 0.9458 - val_loss: 0.1969 - val_accuracy: 0.9444
Epoch 29/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1918 - accuracy: 0.9456 - val_loss: 0.2035 - val_accuracy: 0.9450
Epoch 30/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1919 - accuracy: 0.9458 - val_loss: 0.1999 - val_accuracy: 0.9444
Epoch 31/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1910 - accuracy: 0.9463 - val_loss: 0.1987 - val_accuracy: 0.9439
Epoch 32/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1909 - accuracy: 0.9458 - val_loss: 0.1954 - val_accuracy: 0.9450
Epoch 33/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1902 - accuracy: 0.9461 - val_loss: 0.1986 - val_accuracy: 0.9444
Epoch 34/50
65/65 [==============================] - 0s 3ms/step - loss: 0.1894 - accuracy: 0.9463 - val_loss: 0.1951 - val_accuracy: 0.9444
Epoch 35/50
```

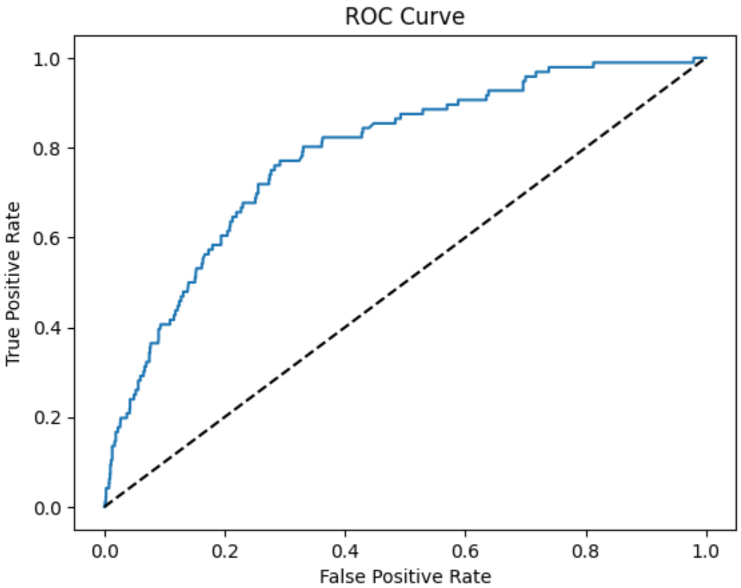
```
# 결과 - accuracy
score1 = l2_model.evaluate(X_train, y_train)
score2 = l2_model.evaluate(X_test, y_test)
print("Training Accuracy: %2f%%₩n" % (score1[1]*100))
print("Test Accuracy: %2f%%₩n" % (score2[1]*100))

# 결과 - recall, f1 score
y_pred_prob = l2_model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print(classification_report(y_test, y_pred))
```

```
130/130 [==============================] - 0s 2ms/step - loss: 0.1829 - accuracy: 0.9449
56/56 [==============================] - 0s 2ms/step - loss: 0.1915 - accuracy: 0.9433
Training Accuracy: 94.487244%
```

```
Test Accuracy: 94.329029%

56/56 [==============================] - 0s 1ms/step
              precision    recall  f1-score   support

         0.0       0.95      0.99      0.97      1685
         1.0       0.31      0.04      0.07        96

    accuracy                           0.94      1781
   macro avg       0.63      0.52      0.52      1781
weighted avg       0.91      0.94      0.92      1781
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = l2_model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
56/56 [==============================] - 0s 3ms/step
```




```
ROC AUC: 0.7856577645895153
<Figure size 640x480 with 0 Axes>
```

```
# L1, L2 규제
l1_l2_model = models.Sequential()
l1_l2_model.add(Dense(8, kernel_regularizer=regularizers.l1_l2(l1=0.01, l2=0.001),
                          activation='relu', input_dim=12))
l1_l2_model.add(Dense(16, kernel_regularizer=regularizers.l1_l2(l1=0.01, l2=0.001),
                          activation='relu'))
l1_l2_model.add(Dense(1, activation='sigmoid'))

l1_l2_model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy',
                 metrics=['accuracy'])


history = l1_l2_model.fit(X_train, y_train, epochs=50, batch_size=64, validation_data=(X_test, y_test))
hist_df = pd.DataFrame(history.history)
hist_df

# y_vloss에 테스트셋의 오차를 저장합니다.
y_vloss = hist_df['val_loss']

# y_loss에 학습셋의 오차를 저장합니다.
y_loss = hist_df['loss']

# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
```

```
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

```
Epoch 1/50
65/65 [==============================] - 1s 6ms/step - loss: 2.5553 - accuracy: 0.4993 - val_loss: 0.8299 - val_accuracy: 0.9461
Epoch 2/50
65/65 [==============================] - 0s 3ms/step - loss: 0.7553 - accuracy: 0.9451 - val_loss: 0.7082 - val_accuracy: 0.9455
Epoch 3/50
65/65 [==============================] - 0s 3ms/step - loss: 0.6616 - accuracy: 0.9451 - val_loss: 0.6251 - val_accuracy: 0.9455
Epoch 4/50
65/65 [==============================] - 0s 3ms/step - loss: 0.5922 - accuracy: 0.9449 - val_loss: 0.5588 - val_accuracy: 0.9461
Epoch 5/50
65/65 [==============================] - 0s 3ms/step - loss: 0.5309 - accuracy: 0.9451 - val_loss: 0.5034 - val_accuracy: 0.9461
Epoch 6/50
65/65 [==============================] - 0s 3ms/step - loss: 0.4804 - accuracy: 0.9451 - val_loss: 0.4564 - val_accuracy: 0.9461
Epoch 7/50
65/65 [==============================] - 0s 3ms/step - loss: 0.4395 - accuracy: 0.9451 - val_loss: 0.4225 - val_accuracy: 0.9461
Epoch 8/50
65/65 [==============================] - 0s 3ms/step - loss: 0.4107 - accuracy: 0.9451 - val_loss: 0.3961 - val_accuracy: 0.9461
Epoch 9/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3850 - accuracy: 0.9451 - val_loss: 0.3749 - val_accuracy: 0.9461
Epoch 10/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3671 - accuracy: 0.9451 - val_loss: 0.3609 - val_accuracy: 0.9461
Epoch 11/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3543 - accuracy: 0.9451 - val_loss: 0.3461 - val_accuracy: 0.9461
Epoch 12/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3397 - accuracy: 0.9451 - val_loss: 0.3335 - val_accuracy: 0.9461
Epoch 13/50
65/65 [==============================] - 0s 4ms/step - loss: 0.3268 - accuracy: 0.9451 - val_loss: 0.3213 - val_accuracy: 0.9461
Epoch 14/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3139 - accuracy: 0.9451 - val_loss: 0.3072 - val_accuracy: 0.9461
Epoch 15/50
65/65 [==============================] - 0s 3ms/step - loss: 0.3032 - accuracy: 0.9451 - val_loss: 0.3012 - val_accuracy: 0.9461
Epoch 16/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2950 - accuracy: 0.9451 - val_loss: 0.2904 - val_accuracy: 0.9461
Epoch 17/50
65/65 [==============================] - 0s 5ms/step - loss: 0.2875 - accuracy: 0.9451 - val_loss: 0.2830 - val_accuracy: 0.9461
Epoch 18/50
65/65 [==============================] - 0s 5ms/step - loss: 0.2815 - accuracy: 0.9451 - val_loss: 0.2806 - val_accuracy: 0.9461
Epoch 19/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2766 - accuracy: 0.9451 - val_loss: 0.2765 - val_accuracy: 0.9461
Epoch 20/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2730 - accuracy: 0.9451 - val_loss: 0.2729 - val_accuracy: 0.9461
Epoch 21/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2695 - accuracy: 0.9451 - val_loss: 0.2701 - val_accuracy: 0.9461
Epoch 22/50
65/65 [==============================] - 0s 4ms/step - loss: 0.2677 - accuracy: 0.9451 - val_loss: 0.2657 - val_accuracy: 0.9461
Epoch 23/50
```

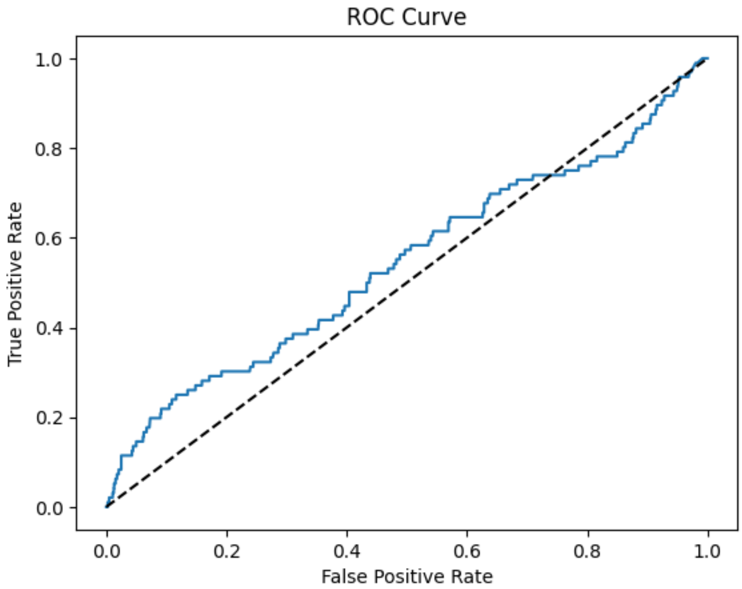
```
# 결과 - accuracy
score1 = l1_l2_model.evaluate(X_train, y_train)
score2 = l1_l2_model.evaluate(X_test, y_test)
print("Training Accuracy: %2f%%₩n" % (score1[1]*100))
print("Test Accuracy: %2f%%₩n" % (score2[1]*100))

# 결과 - recall, f1 score
y_pred_prob = l1_l2_model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print(classification_report(y_test, y_pred))
```

```
130/130 [==============================] - 0s 2ms/step - loss: 0.2360 - accuracy: 0.9451
56/56 [==============================] - 0s 2ms/step - loss: 0.2364 - accuracy: 0.9461
Training Accuracy: 94.511312%

Test Accuracy: 94.609767%

56/56 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

         0.0       0.95      1.00      0.97      1685
         1.0       0.00      0.00      0.00        96

    accuracy                           0.95      1781
   macro avg       0.47      0.50      0.49      1781
weighted avg       0.90      0.95      0.92      1781

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = l1_l2_model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
```

```
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

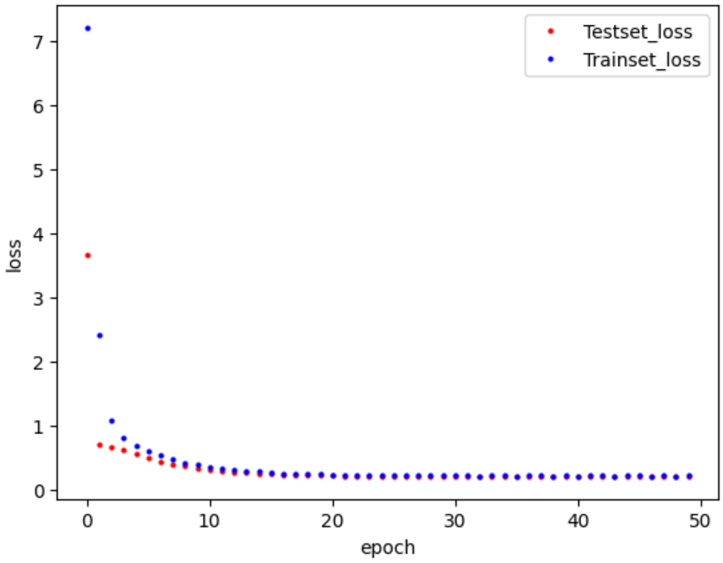
```
56/56 [==============================] - 0s 3ms/step
```




```
ROC AUC: 0.542328140454995
<Figure size 640x480 with 0 Axes>
```

```
# dropout 규제 적용
from tensorflow.keras.layers import Dropout
dpt_model = models.Sequential()
dpt_model.add(Dense(8, kernel_regularizer=regularizers.l1(0.01), activation='relu', input_dim=12))
# Dense로 촘촘히 그리고 dropout으로 한칸씩 빼고..(정보를 50%씩 빼는거임)
# 즉 dropout은 엄청난 오버피팅이 일어났을 때 쓰는 방법으로 L1,L2보다 훨씬 강한 규제임
# 결론적으로, 일단은 오버피팅이 일어나도 좋으니 학습횟수를 크게 늘린 후(제일 잘 맞추는 모델을 생성한 후)에 오버피팅이 발생하면 규제를 적용해야함
dpt_model.add(Dropout(0.5))
dpt_model.add(Dense(16, kernel_regularizer=regularizers.l1(0.01), activation='relu'))
dpt_model.add(Dropout(0.5))
dpt_model.add(Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])


history = dpt_model.fit(X_train, y_train, epochs=50, batch_size=64, validation_data=(X_test, y_test))
hist_df = pd.DataFrame(history.history)
hist_df

# y_vloss에 테스트셋의 오차를 저장합니다.
y_vloss = hist_df['val_loss']

# y_loss에 학습셋의 오차를 저장합니다.
y_loss = hist_df['loss']

# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

```
65/65 [==============================] - 0s 6ms/step - loss: 0.2220 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 36/50
65/65 [==============================] - 1s 9ms/step - loss: 0.2192 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 37/50
65/65 [==============================] - 1s 11ms/step - loss: 0.2210 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 38/50
65/65 [==============================] - 1s 12ms/step - loss: 0.2231 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 39/50
65/65 [==============================] - 0s 7ms/step - loss: 0.2193 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 40/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2204 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 41/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2190 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 42/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2206 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 43/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2213 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 44/50
65/65 [==============================] - 0s 8ms/step - loss: 0.2179 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 45/50
65/65 [==============================] - 0s 5ms/step - loss: 0.2232 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 46/50
65/65 [==============================] - 0s 6ms/step - loss: 0.2206 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 47/50
65/65 [==============================] - 0s 5ms/step - loss: 0.2188 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 48/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2201 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
Epoch 49/50
65/65 [==============================] - 0s 7ms/step - loss: 0.2178 - accuracy: 0.9451 - val_loss: 0.2111 - val_accuracy: 0.9461
Epoch 50/50
65/65 [==============================] - 1s 8ms/step - loss: 0.2199 - accuracy: 0.9451 - val_loss: 0.2110 - val_accuracy: 0.9461
```

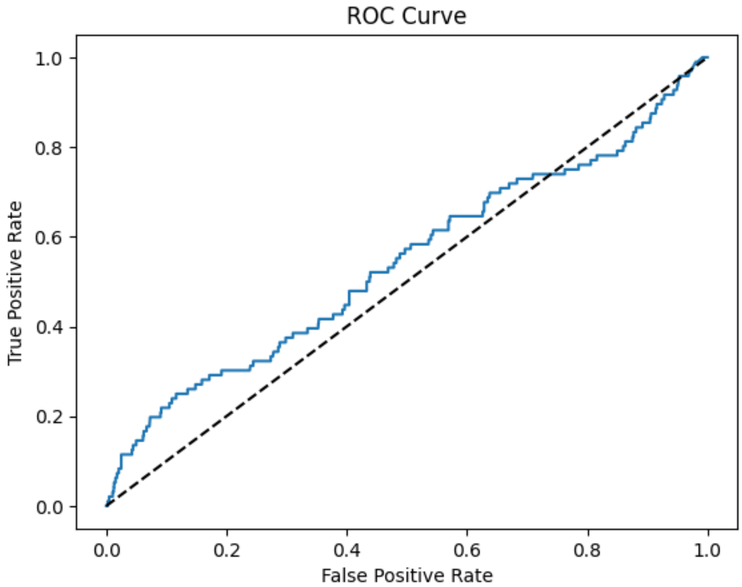


```
# 결과 - accuracy
score1 = dpt_model.evaluate(X_train, y_train)
score2 = dpt_model.evaluate(X_test, y_test)
print("Training Accuracy: %2f%%Wn" % (score1[1]*100))
print("Test Accuracy: %2f%%Wn" % (score2[1]*100))

# 결과 - recall, f1 score
y_pred_prob = dpt_model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print(classification_report(y_test, y_pred))
```

```
130/130 [==============================] - 0s 2ms/step - loss: 0.2138 - accuracy: 0.9451
56/56 [==============================] - 0s 2ms/step - loss: 0.2110 - accuracy: 0.9461
Training Accuracy: 94.511312%

Test Accuracy: 94.609767%

56/56 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

         0.0       0.95      1.00      0.97      1685
         1.0       0.00      0.00      0.00        96

    accuracy                           0.95      1781
   macro avg       0.47      0.50      0.49      1781
weighted avg       0.90      0.95      0.92      1781

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = l1_l2_model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black') #diagonal line
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
56/56 [==============================] - 0s 3ms/step
```



SMOTE 알고리즘을 활용해 데이터 확장

```
# SMOTE 사용
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=0)
X_train_over, y_train_over = smote.fit_resample(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())
```

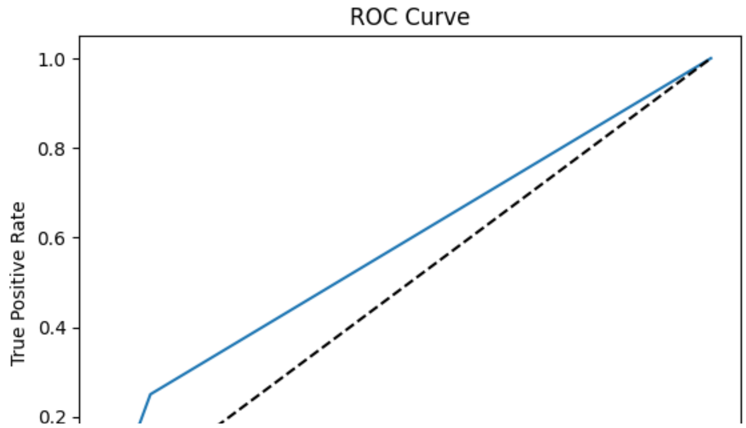
```
SMOTE 적용 전 학습용 피처/레이블 데이터 세트:  (4154, 12) (4154,)
SMOTE 적용 후 학습용 피처/레이블 데이터 세트:  (7852, 12) (7852,)
SMOTE 적용 후 레이블 값 분포:
 0.0    3926
1.0    3926
dtype: int64
```

```
# 최적의사결정트리: pre-prunning max_depth = 3
tree1 = DecisionTreeClassifier(max_depth=3, random_state=0)
tree1.fit(X_train_over, y_train_over)

print("Accuracy on training set: {:.3f}".format(tree1.score(X_train_over, y_train_over)))
print("Accuracy on test set: {:.3f}".format(tree1.score(X_test, y_test)))
```

```
Accuracy on training set: 0.826
Accuracy on test set: 0.895
```

```
y_pred = tree1.predict(X_test)
print(classification_report(y_test, y_pred, target_names=['class 0', 'class 1']))
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = tree1.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
              precision    recall  f1-score   support

     class 0       0.96      0.93      0.94      1685
     class 1       0.17      0.25      0.20        96

    accuracy                           0.90      1781
   macro avg       0.56      0.59      0.57      1781
weighted avg       0.91      0.90      0.90      1781
```




```
#최적의 랜덤포레스트
forest3 = RandomForestClassifier(n_estimators=100, max_depth=10, class_weight=class_weights, random_state=0)
forest3.fit(X_train_over, y_train_over)

print("Accuracy on training set: {:.3f}".format(forest3.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest3.score(X_test, y_test)))
```

```
Accuracy on training set: 0.729
Accuracy on test set: 0.712
```

```
y_pred3 = forest3.predict(X_test)
print(classification_report(y_test, y_pred3, target_names=['class 0', 'class 1']))

# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = tree1.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```

```
                  precision    recall  f1-score   support

         class 0       0.98      0.71      0.82      1685
         class 1       0.12      0.72      0.21        96

        accuracy                           0.71      1781
       macro avg       0.55      0.72      0.52      1781
    weighted avg       0.93      0.71      0.79      1781
```

```
xgbest_model = xgb.XGBClassifier()

# 그리드 서치
param_grid = {
    'max_depth': [3, 4, 5],
    'learning_rate': [0.1, 0.01, 0.001],
    'n_estimators': [100, 200, 300],
}

grid_search = GridSearchCV(estimator=xgbest_model, param_grid=param_grid, scoring='accuracy', cv=3)
grid_search.fit(X_train_over, y_train_over)

# 최적의 하이퍼파라미터를 출력
print("Best Parameters: ", grid_search.best_params_)
# 최적 모델로 테스트 데이터로 예측
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
# 정확도 출력
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy on test set: {:.3f}".format(accuracy))

y_pred = best_model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=['class 0', 'class 1']))
```

```
Best Parameters:  {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200}
Accuracy on test set: 0.936
              precision    recall  f1-score   support

     class 0       0.95      0.98      0.97      1685
     class 1       0.28      0.11      0.16        96

    accuracy                           0.94      1781
   macro avg       0.61      0.55      0.56      1781
weighted avg       0.91      0.94      0.92      1781
```

```
# 결과 - ROC 곡선
from sklearn.metrics import roc_curve, roc_auc_score
y_test_pred_probs = best_model.predict(X_test)
FPR, TPR, _ = roc_curve(y_test, y_test_pred_probs)
plt.plot(FPR, TPR)
plt.plot([0,1],[0,1],'--', color='black')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
plt.clf()
# AUC 계산
auc = roc_auc_score(y_test, y_test_pred_probs)
print("ROC AUC:", auc)
```