**패키지**

import numpy as np

import pandas as pd

import json

import datetime as dt

import matplotlib.pyplot as plt

import matplotlib.ticker as ticker

get\_ipython().run\_line\_magic('matplotlib', 'inline')

import seaborn as sns

plt.style.use('seaborn-whitegrid')

import re

import glob

import os

from scipy import stats

from scipy.integrate import trapz

import missingno as msno

import sys

import warnings

warnings.filterwarnings('ignore')

pd.set\_option('max\_columns', 20, 'max\_rows', 20, 'max\_colwidth', 10)

# 데이터 분리

from sklearn.model\_selection import train\_test\_split, StratifiedKFold, KFold

from sklearn.model\_selection import cross\_val\_score, GridSearchCV, RepeatedStratifiedKFold

# 전처리

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.preprocessing import Binarizer

# 머신러닝 알고리즘

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from xgboost import XGBClassifier

from lightgbm import LGBMClassifier

from sklearn.gaussian\_process import GaussianProcessClassifier

from sklearn.naive\_bayes import GaussianNB

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.ensemble import ExtraTreesClassifier

from sklearn.ensemble import AdaBoostClassifier

from sklearn.ensemble import BaggingClassifier

from sklearn.ensemble import VotingClassifier

# 모델 튜닝 및 평가

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import GridSearchCV

from sklearn.model\_selection import cross\_val\_predict

from sklearn import model\_selection

from sklearn.metrics import make\_scorer

# 평가 패키지

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, roc\_auc\_score

from sklearn.metrics import f1\_score, confusion\_matrix, precision\_recall\_curve, roc\_curve

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

# # train과 test가 나눠져 있지 않은 경우 아래처럼 train과 test를 나눈다.

# X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=156, stratify=y )

train = train\_df.copy()

test = test\_df.copy()

# 병합 준비

ntrain = train.shape[0]

ntest = test.shape[0]

# 아래는 따로 잘 모셔 둡니다.

y\_train = train['Survived'].values

# 병함 파일 만들기

data = pd.concat((train, test))

# 피쳐엔지니어링 예시

temp = data.copy()

temp['Initial'] = 0

temp['Initial'] = data.Name.str.extract('([A-Za-z0-9]+)\.')

temp.loc[temp['Initial'] == 'Dona', 'Initial'] = 'Mrs'

temp['LastName'] = data.Name.str.extract('([A-Za-z]+)')

temp['NumName'] = temp['LastName'].factorize()[0]

temp = temp.reset\_index(drop=True)

temp['Age'] = temp.groupby('Initial')['Age'].apply(lambda x: x.fillna(x.mean()))

temp['Initial'].replace(['Capt', 'Col', 'Countess', 'Don', 'Dona' , 'Dr', 'Jonkheer', 'Lady', 'Major', 'Master', 'Miss' ,'Mlle', 'Mme', 'Mr', 'Mrs', 'Ms', 'Rev', 'Sir'], ['Sacrificed', 'Respected', 'Nobles', 'Mr', 'Mrs', 'Respected', 'Mr', 'Nobles', 'Respected', 'Kids', 'Miss', 'Nobles', 'Nobles', 'Mr', 'Mrs', 'Nobles', 'Sacrificed', 'Nobles'],inplace=True)

temp['Initial'].replace(['Kids', 'Miss', 'Mr', 'Mrs', 'Nobles', 'Respected', 'Sacrificed'], [4, 4, 2, 5, 6, 3, 1], inplace=True)

temp['Age\_Range'] = pd.qcut(temp['Age'], 10)

temp['Agroup'] = 0

temp.loc[temp['Age'] < 1.0, 'Agroup'] = 1

temp.loc[(temp['Age'] >=1.0) & (temp['Age'] <= 3.0), 'Agroup'] = 2

temp.loc[(temp['Age'] > 3.0) & (temp['Age'] < 11.0), 'Agroup'] = 7

temp.loc[(temp['Age'] >= 11.0) & (temp['Age'] < 15.0), 'Agroup'] = 13

temp.loc[(temp['Age'] >= 15.0) & (temp['Age'] < 18.0), 'Agroup'] = 16

temp.loc[(temp['Age'] >= 18.0) & (temp['Age'] <= 20.0), 'Agroup'] = 18

temp.loc[(temp['Age'] > 20.0) & (temp['Age'] <= 22.0), 'Agroup'] = 21

temp.loc[(temp['Age'] > 22.0) & (temp['Age'] <= 26.0), 'Agroup'] = 24

temp.loc[(temp['Age'] > 26.0) & (temp['Age'] <= 30.0), 'Agroup'] = 28

temp.loc[(temp['Age'] > 30.0) & (temp['Age'] <= 32.0), 'Agroup'] = 31

temp.loc[(temp['Age'] > 32.0) & (temp['Age'] <= 34.0), 'Agroup'] = 33

temp.loc[(temp['Age'] > 34.0) & (temp['Age'] <= 38.0), 'Agroup'] = 36

temp.loc[(temp['Age'] > 38.0) & (temp['Age'] <= 52.0), 'Agroup'] = 45

temp.loc[(temp['Age'] > 52.0) & (temp['Age'] <= 75.0), 'Agroup'] = 60

temp.loc[temp['Age'] > 75.0, 'Agroup'] = 78

temp.loc[(temp['Sex'] == 'male'), 'Sex'] = 1

temp.loc[(temp['Sex'] == 'female'), 'Sex'] = 2

temp.loc[(temp['Age'] < 1), 'Sex'] = 3

temp.loc[(temp['SibSp'] == 0) & (temp['Parch'] == 0), 'Alone'] = 1

temp['Family'] = temp['Parch'] + temp['SibSp'] + 1

temp['Initick'] = temp.Ticket.str.extract('^([A-Za-z0-9]+)')

temp = temp.reset\_index(drop=True)

temp['Initick'] = temp.Ticket.str.extract('^([A-Za-z]+)')

temp['NumTicket'] = temp['Initick'].factorize()[0]

temp['Fare\_Range'] = pd.qcut(train['Fare'], 10)

temp['Fgroup'] = 0

temp.loc[temp['Fare'] <= 0,'Fgroup'] = 0

temp.loc[(temp['Fare'] > 0) & (temp['Fare'] <= 7.125), 'Fgroup'] = 1

temp.loc[(temp['Fare'] > 7.125) & (temp['Fare'] <= 7.9), 'Fgroup'] = 2

temp.loc[(temp['Fare'] > 7.9) & (temp['Fare'] <= 8.03), 'Fgroup'] = 3

temp.loc[(temp['Fare'] > 8.03) & (temp['Fare'] < 10.5), 'Fgroup'] = 4

temp.loc[(temp['Fare'] >= 10.5) & (temp['Fare'] < 23.0), 'Fgroup'] = 5

temp.loc[(temp['Fare'] >= 23.0) & (temp['Fare'] <= 27.8), 'Fgroup'] = 6

temp.loc[(temp['Fare'] > 27.8) & (temp['Fare'] <= 51.0), 'Fgroup'] = 7

temp.loc[(temp['Fare'] > 51.0) & (temp['Fare'] <= 73.5), 'Fgroup'] = 8

temp.loc[temp['Fare'] > 73.5, 'Fgroup'] = 9

temp['Inicab'] = 0

temp['Inicab'] = temp['Cabin'].str.extract('^([A-Za-z]+)')

temp.loc[((temp['Cabin'].isnull()) & (temp['Pclass'].values == 1)), 'Inicab'] = 'X'

temp.loc[((temp['Cabin'].isnull()) & (temp['Pclass'].values == 2)), 'Inicab'] = 'Y'

temp.loc[((temp['Cabin'].isnull()) & (temp['Pclass'].values == 3)), 'Inicab'] = 'Z'

temp['Inicab'] = temp['Inicab'].factorize()[0]

temp.loc[(temp.Embarked.isnull()), 'Embarked'] = 'S'

temp['Embarked'] = temp['Embarked'].factorize()[0]

temp['Priority'] = 0

temp.loc[(temp['Initial'] == 6), 'Priority'] = 1

temp.loc[(temp['Pclass'] == 1) & (temp['Sex'] == 2), 'Priority'] = 2

temp.loc[(temp['Age'] < 1), 'Priority'] = 3

temp.loc[(temp['Pclass'] == 1) & (temp['Age'] <= 17), 'Priority'] = 4

temp.loc[(temp['Pclass'] == 2) & (temp['Age'] <= 17), 'Priority'] = 5

temp.loc[(temp['Pclass'] == 2) & (temp['Sex'] == 2), 'Priority'] = 6

temp.loc[(temp['Fgroup'] == 9), 'Priority'] = 7

temp['FH'] = 0

temp.loc[(temp['Sex'] == 1), 'FH'] = 0

temp.loc[(temp['Sex'] == 2), 'FH'] = 1

temp.loc[(temp['Sex'] == 2) & (temp['Family'] == 2), 'FH'] = 2

temp.loc[(temp['Sex'] == 2) & (temp['Family'] == 3), 'FH'] = 3

temp.loc[(temp['Sex'] == 2) & (temp['Family'] == 4), 'FH'] = 4

temp.loc[(temp['Sex'] == 2) & (temp['Family'] == 1) & (temp['Pclass'] == 1), 'FH'] = 5

temp.loc[(temp['Sex'] == 2) & (temp['Family'] == 1) & (temp['Pclass'] == 2), 'FH'] = 6

temp.loc[(temp['Sex'] == 2) & (temp['Fgroup'] == 3), 'FH'] = 7

temp.loc[(temp['Sex'] == 2) & (temp['Fgroup'] >= 5), 'FH'] = 8

temp['MH'] = 0

temp.loc[(temp['Sex'] == 2), 'MH'] = 0

temp.loc[(temp['Sex'] == 1), 'MH'] = 1

temp.loc[(temp['Sex'] == 1) & (temp['Family'] == 2), 'MH'] = 2

temp.loc[(temp['Sex'] == 1) & (temp['Family'] == 3), 'MH'] = 3

temp.loc[(temp['Sex'] == 1) & (temp['Family'] == 4), 'MH'] = 4

temp.loc[(temp['Sex'] == 1) & (temp['Family'] == 1) & (temp['Pclass'] == 1), 'MH'] = 5

temp.loc[(temp['Sex'] == 1) & (temp['Family'] == 1) & (temp['Pclass'] == 2), 'MH'] = 6

temp.loc[(temp['Sex'] == 1) & (temp['Fgroup'] == 3), 'MH'] = 7

temp.loc[(temp['Sex'] == 1) & (temp['Fgroup'] >= 5), 'MH'] = 8

temp['FL'] = 0

temp.loc[(temp['Sex'] == 1), 'FL'] = 0

temp.loc[(temp['Sex'] == 2) & (temp['Fgroup'] < 5), 'FL'] = 1

temp.loc[(temp['Sex'] == 2) & (temp['Fgroup'] != 3), 'FL'] = 2

temp.loc[(temp['Sex'] == 2) & (temp['FH'] == 1), 'FL'] = 3

temp.loc[(temp['Sex'] == 2) & (temp['Family'] < 2), 'FL'] = 4

temp.loc[(temp['Sex'] == 2) & (temp['Family'] > 4), 'FL'] = 5

temp.loc[(temp['Sex'] == 2) & (temp['Family'] == 1) & (temp['Pclass'] == 3), 'FL'] = 6

temp['ML'] = 0

temp.loc[(temp['Sex'] == 2), 'ML'] = 0

temp.loc[(temp['Sex'] == 1) & (temp['Fgroup'] < 5), 'ML'] = 1

temp.loc[(temp['Sex'] == 1) & (temp['Fgroup'] != 3), 'ML'] = 2

temp.loc[(temp['Sex'] == 1) & (temp['MH'] <7), 'ML'] = 3

temp.loc[(temp['Sex'] == 1) & (temp['Family'] < 2), 'ML'] = 4

temp.loc[(temp['Sex'] == 1) & (temp['Family'] > 4), 'ML'] = 5

temp.loc[(temp['Sex'] == 1) & (temp['Family'] == 1) & (temp['Pclass'] == 3), 'ML'] = 6

temp['F1'] = temp['Priority']

temp['F2'] = temp['FH']

temp['F3'] = temp['MH']

temp['F4'] = temp['FL']

temp['F5'] = temp['ML']

temp['F6'] = temp['Initial']

temp['F7'] = temp['Fgroup']

temp['F8'] = temp['NumName']

temp['F9'] = temp['NumTicket']

temp['F10'] = temp['Family']

temp['F11'] = temp['Embarked']

temp['F12'] = temp['Sex']

temp['F13'] = temp['Pclass']

dfl = pd.DataFrame() # for label encoding

good\_columns = ['F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9', 'F10', 'F11', 'F12', 'F13']

dfl[good\_columns] = temp[good\_columns]

dfl\_enc = dfl.apply(LabelEncoder().fit\_transform)

dfh = dfl.copy()

one\_hot\_cols = dfh.columns.tolist()

dfh\_enc = pd.get\_dummies(dfh, columns=one\_hot\_cols)

dfh\_enc

train = dfh\_enc[:ntrain]

test = dfh\_enc[ntrain:]

X\_test = test

X\_train = train

**# # 데이터 확인**

**# ## y값 정규성 확인**

y\_target

sns.distplot(y\_target)

sns.distplot(log1p(y\_target))

sns.distplot(Stan)

**# ## x값 선형성 확인**

# 2개의 행과 4개의 열을 가진 subplots를 이용. axs는 4x2개의 ax를 가짐.

fig, axs = plt.subplots(figsize=(16,16) , ncols=4 , nrows=4)

features = ['season', 'holiday', 'workingday', 'weather', 'temp', 'atemp',

'humidity', 'windspeed', 'casual', 'registered', 'year',

'month', 'day', 'hour']

for i , feature in enumerate(features):

row = int(i/4)

col = i%4

# 시본의 regplot을 이용해 산점도와 선형 회귀 직선을 함께 표현

sns.regplot(x=feature , y='count',data=train\_1 , ax=axs[row][col])

**# # 전처리**

**# ## 데이터 분리**

# 전처리가 모두 끝나고 scaling은 하지 않은 상태에서 데이터 분리를 시행한다.

X\_features = diabetes\_df.iloc[:, :-1]

y\_target = diabetes\_df.iloc[:, -1]

X\_features.shape[0]/diabetes\_df.shape[0]

y\_target.shape[0]/diabetes\_df.shape[0]

y\_target.value\_counts()/y\_target.shape[0]

# ### train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_features, y\_target, test\_size = 0.2, random\_state = 156, stratify=y\_target)

print('학습 세트 Shape:{0}, 테스트 세트 Shape:{1}'.format(X\_train.shape , X\_test.shape))

print(' 학습 세트 레이블 값 분포 비율')

print(y\_train.value\_counts()/y\_train.count())

print('\n 테스트 세트 레이블 값 분포 비율')

print(y\_test.value\_counts()/y\_test.count())

# ### K-fold(전처리 후 진행해야 함)

# - 1,2,3,4 폴드로 한번씩 나눠서 뿌리기 때문에 예측도 같이 돌려야 함

dt\_clf = DecisionTreeClassifier(random\_state=156)

# 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체 생성.

kfold = KFold(n\_splits=5)

n\_iter = 0

cv\_accuracy = []

# KFold객체의 split( ) 호출하면 폴드 별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환

for train\_index, test\_index in kfold.split(X\_features):

# kfold.split( )으로 반환된 인덱스를 이용하여 학습용, 검증용 테스트 데이터 추출

X\_train, X\_test = X\_features[train\_index], X\_features[test\_index]

y\_train, y\_test = y\_target[train\_index], y\_target[test\_index]

#학습 및 예측

dt\_clf.fit(X\_train , y\_train)

pred = dt\_clf.predict(X\_test)

n\_iter += 1

# 반복 시 마다 정확도 측정

accuracy = np.round(accuracy\_score(y\_test,pred), 4)

train\_size = X\_train.shape[0]

test\_size = X\_test.shape[0]

print('\n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'

.format(n\_iter, accuracy, train\_size, test\_size))

print('#{0} 검증 세트 인덱스:{1}'.format(n\_iter,test\_index))

cv\_accuracy.append(accuracy)

# 개별 iteration별 정확도를 합하여 평균 정확도 계산

print('\n## 평균 검증 정확도:', np.mean(cv\_accuracy))

# ### Stratified K-fold(전처리 후 진행해야 함)

dt\_clf = DecisionTreeClassifier(random\_state=156)

skfold = StratifiedKFold(n\_splits=3)

n\_iter=0

cv\_accuracy=[]

# StratifiedKFold의 split( ) 호출시 반드시 레이블 데이터 셋도 추가 입력 필요

for train\_index, test\_index in skfold.split(X\_features, y\_target):

# split( )으로 반환된 인덱스를 이용하여 학습용, 검증용 테스트 데이터 추출

X\_train, X\_test = X\_features[train\_index], X\_features[test\_index]

y\_train, y\_test = y\_target[train\_index], y\_target[test\_index]

#학습 및 예측

dt\_clf.fit(X\_train , y\_train)

pred = dt\_clf.predict(X\_test)

# 반복 시 마다 정확도 측정

n\_iter += 1

accuracy = np.round(accuracy\_score(y\_test,pred), 4)

train\_size = X\_train.shape[0]

test\_size = X\_test.shape[0]

print('\n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'

.format(n\_iter, accuracy, train\_size, test\_size))

print('#{0} 검증 세트 인덱스:{1}'.format(n\_iter,test\_index))

cv\_accuracy.append(accuracy)

# 교차 검증별 정확도 및 평균 정확도 계산

print('\n## 교차 검증별 정확도:', np.round(cv\_accuracy, 4))

print('## 평균 검증 정확도:', np.mean(cv\_accuracy))

# ### SMOTE 오버 샘플링 적용 후 모델 학습/예측/평가

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())

**# ## 범주 데이터 수치화**

# ### get\_dummies

col\_list = ['TV','냉장고','전자렌지','컴퓨터','선풍기','선풍기','믹서','믹서']

col\_ohe = pd.get\_dummies(X\_train[col\_list])

X\_train\_ohe = pd.concat([X\_train, col\_ohe], axis=1)

X\_train = X\_train\_ohe.drop(col\_list, axis=1)

# ### Lable Incoding(미완)

from sklearn.preprocessing import LabelEncoder

col\_list=['TV','냉장고','전자렌지','컴퓨터','선풍기','선풍기','믹서','믹서']

# LabelEncoder를 객체로 생성한 후 , fit( ) 과 transform( ) 으로 label 인코딩 수행.

encoder = LabelEncoder()

encoder.fit(col\_list)

labels = encoder.transform(col\_list)

print('인코딩 변환값:',labels)

print('인코딩 클래스:',encoder.classes\_)

print('디코딩 원본 값:',encoder.inverse\_transform([4, 5, 2, 0, 1, 1, 3, 3]))

# ### one-hot Incoding(미완)

from sklearn.preprocessing import OneHotEncoder

import numpy as np

col\_list=['TV','냉장고','전자렌지','컴퓨터','선풍기','선풍기','믹서','믹서']

# 먼저 숫자값으로 변환을 위해 LabelEncoder로 변환합니다.

encoder = LabelEncoder()

encoder.fit(col\_list)

labels = encoder.transform(col\_list)

# 2차원 데이터로 변환합니다.

labels = labels.reshape(-1,1)

# 원-핫 인코딩을 적용합니다.

oh\_encoder = OneHotEncoder()

oh\_encoder.fit(labels)

oh\_labels = oh\_encoder.transform(labels)

print('원-핫 인코딩 데이터')

print(oh\_labels.toarray())

print('원-핫 인코딩 데이터 차원')

print(oh\_labels.shape)

**# ## Scaler**

\* y부터 스케일링 시작

\* X의 경우 중요한 feature를 중심으로 골라 분포를 확인 후 스케일링

\* X\_train을 fit 시키고 X\_test에 transform하는 경우가 있음

\* 2차원만 스케일링이 가능하므로 DataFrame 형태만 처리 됨

# ### StandardScaler

# X\_features scaling

scaler = StandardScaler()

scaler = scaler.fit(X\_train)

X\_train\_S = pd.DataFrame(scaler.transform(X\_train),

index=X\_train.index, columns=X\_train.columns)

X\_test\_S = pd.DataFrame(scaler.transform(X\_test),

index=X\_test.index, columns=X\_test.columns)

display(X\_test\_S.describe().T)

# y\_target scaling

scaler = StandardScaler()

scaler = scaler.fit(y\_train)

y\_train\_S = pd.DataFrame(scaler.transform(y\_train),

index=y\_train.index, columns=y\_train.columns)

y\_test\_S = pd.DataFrame(scaler.transform(y\_test),

index=y\_test.index, columns=y\_test.columns)

display(y\_test\_S.describe().T)

# ### MinMax

# X\_features scaling

scaler = MinMaxScaler()

scaler = scaler.fit(X\_train)

X\_train\_S = pd.DataFrame(scaler.transform(X\_train),

index=X\_train.index, columns=X\_train.columns)

X\_test\_S = pd.DataFrame(scaler.transform(X\_test),

index=X\_test.index, columns=X\_test.columns)

display(X\_test\_S.describe().T)

# y\_target scaling

scaler = MinMaxScaler()

scaler = scaler.fit(y\_train)

y\_train\_S = pd.DataFrame(scaler.transform(y\_train),

index=y\_train.index, columns=y\_train.columns)

y\_test\_S = pd.DataFrame(scaler.transform(y\_test),

index=y\_test.index, columns=y\_test.columns)

display(y\_test\_S.describe().T)

# ### Normalizer()

# X\_features scaling

scaler = Normalizer()

scaler = scaler.fit(X\_train)

X\_train\_S = pd.DataFrame(scaler.transform(X\_train),

index=X\_train.index, columns=X\_train.columns)

X\_test\_S = pd.DataFrame(scaler.transform(X\_test),

index=X\_test.index, columns=X\_test.columns)

display(X\_test\_S.describe().T)

# y\_target scaling

scaler = Normalizer()

scaler = scaler.fit(y\_train)

y\_train\_S = pd.DataFrame(scaler.transform(y\_train),

index=y\_train.index, columns=y\_train.columns)

y\_test\_S = pd.DataFrame(scaler.transform(y\_test),

index=y\_test.index, columns=y\_test.columns)

display(y\_test\_S.describe().T)

# ### Log1p

# X\_features scaling

X\_train\_S = pd.DataFrame(np.log1p(X\_train), index=X\_train.index, columns=X\_train.columns)

X\_test\_S = pd.DataFrame(np.log1p(X\_test), index=X\_test.index, columns=X\_test.columns)

display(X\_test\_S.describe().T)

# y\_target scaling

y\_train\_S = pd.DataFrame(np.log1p(y\_train), index=X\_train.index, columns=y\_train.columns)

y\_test\_S = pd.DataFrame(np.log1p(y\_test), index=y\_test.index, columns=y\_test.columns)

display(y\_test\_S.describe().T)

**# ## 정규성 확인**

sns.distplot(house\_df['SalePrice'])

**# ## 다중공선성제거**

# ### VIF

# #### 상관도 확인

corr = X\_features.corr()

plt.figure(figsize=(14,14))

sns.heatmap(corr, annot=True, fmt='.1g')

# correlation from features

raw\_df.corr().loc[X\_colname, X\_colname].style.background\_gradient().set\_precision(2).set\_properties(\*\*{'font-size': '11pt'})

corr = X\_features.corr().loc[:, ['count']]

corr = X\_features.corr().loc[:, ['count']].style.background\_gradient().set\_precision(2).set\_properties(\*\*{'font-size': '11pt'})

corr

raw\_df.describe()

for col in raw\_df.describe().columns:

target = raw\_df[col]

figure, axes = plt.subplots(2,1,figsize=(16,10))

sm.graphics.tsa.plot\_acf(target, lags=100, use\_vlines=True, ax=axes[0], title=col)

sm.graphics.tsa.plot\_pacf(target, lags=100, use\_vlines=True, ax=axes[1], title=col)

vif = pd.DataFrame()

vif['VIF\_Factor'] = [variance\_inflation\_factor(X\_train\_S.values, i)

for i in range(X\_train\_S.shape[1])]

vif['Feature'] = X\_train\_S.columns

vif.sort\_values(by='VIF\_Factor', ascending=True)

# extract effective features using variance inflation factor

vif = pd.DataFrame()

vif['VIF\_Factor'] = [variance\_inflation\_factor(X\_train\_S.values, i)

for i in range(X\_train\_S.shape[1])]

vif['Feature'] = X\_train\_S.columns

vif.sort\_values(by='VIF\_Factor', ascending=True)['Feature'][:10].values

# #### 독립변수 일부 반영

### Functionalize

### extract non-multicollinearity variables by VIF

def feature\_engineering\_XbyVIF(X\_train, num\_variables):

vif = pd.DataFrame()

vif['VIF\_Factor'] = [variance\_inflation\_factor(X\_train.values, i)

for i in range(X\_train.shape[1])]

vif['Feature'] = X\_train.columns

X\_colname\_vif = vif.sort\_values(by='VIF\_Factor', ascending=True)['Feature'][:num\_variables].values

return X\_colname\_vif

# X\_colname\_vif = feature\_engineer

feature\_engineering\_XbyVIF(X\_train\_S, 10)

# ### PCA

pca = PCA(n\_components=2)

#fit( )과 transform( ) 을 호출하여 PCA 변환 데이터 반환

pca.fit(X\_features)

X\_features\_pca = pca.transform(X\_features)

print('PCA Component별 변동성:', pca.explained\_variance\_ratio\_)

# ### LDA

lda = LinearDiscriminantAnalysis(n\_components=2)

# fit()호출 시 target값 입력

lda.fit(X\_features, y\_target)

X\_features\_lda = lda.transform(X\_features)

print(iris\_lda.shape)

lda\_columns=['lda\_component\_1','lda\_component\_2']

X\_lda\_df = pd.DataFrame(X\_features\_lda,columns=lda\_columns)

X\_lda\_df['target']=y\_target

# ### SVD

# ### Truncated SVD

# ### NMF

**# # 머신러닝 모델 만들기**

dt\_clf = DecisionTreeClassifier()

rf\_clf = RandomForestClassifier(random\_state=1)

knn\_clf = KNeighborsClassifier()

lr\_clf = LogisticRegression()

xgb\_clf = XGBClassifier()

lgbm\_clf = LGBMClassifier()

gb\_clf = GradientBoostingClassifier()

svc\_clf = SVC(probability=True)

ext\_clf = ExtraTreesClassifier()

ada\_clf = AdaBoostClassifier()

gnb\_clf = GaussianNB()

gp\_clf = GaussianProcessClassifier()

bag\_clf = BaggingClassifier()

# 리스트 준비

models = [dt\_clf, rf\_clf, knn\_clf, lr\_clf, xgb\_clf, lgbm\_clf, gb\_clf, svc\_clf, ext\_clf, ada\_clf, gnb\_clf, gp\_clf, bag\_clf]

model\_names = ['Decition Tree', 'Random Forest', 'K Nearest Neighbour', 'Logistic Regression', 'XGBoost', 'LightGBM', 'Gradient Boosting', 'SVC', 'Extra Trees', 'AdaBoost', 'Gaussian Naive Bayes', 'Gaussian Process', 'Bagging Classifier']

scores = {}

# 이어서 연속적으로 모델을 학습 시키고 교차 검증합니다.

for ind, mod in enumerate(models):

mod.fit(X\_train, y\_train)

acc = cross\_val\_score(mod, X\_train, y\_train, scoring = "accuracy", cv = 10)

scores[model\_names[ind]] = acc

**# ## 결과 : 테이블로 보기**

# 결과 테이블을 만듭니다.

results = pd.DataFrame(scores).T

results['mean'] = results.mean(1)

result\_df = results.sort\_values(by='mean', ascending=False)#.reset\_index()

result\_df

**# ## 결과 : 그래프(boxplot)으로 보기**

result\_df = result\_df.drop(['mean'], axis=1)

sns.boxplot(data=result\_df.T, orient='h')

plt.title('Machine Learning Algorithm Accuracy Score \n')

plt.xlabel('Accuracy Score (%)');

**# # 중요도에 따라 모델 재설정**

**# ## 중요도 그래프로 보기**

# 중요도를 보는 함수를 만듭니다.

def importance\_plotting(data, xlabel, ylabel, title, n=20):

sns.set(style="whitegrid")

ax = data.tail(n).plot(kind='barh')

ax.set(title=title, xlabel=xlabel, ylabel=ylabel)

ax.xaxis.grid(False)

ax.yaxis.grid(True)

plt.show()

# 중요도를 데이터프레임에 넣습니다. Logistic regression에서는 중요도보다 coefficients를 사용합니다.

# 아래는 Features라는 열에 트레인의 열들의 이름을 리스트로 만들어서 넣고 Importance에는 Logistic regression에는 coefficient를 바꾸어 넣어라는 넘파이 명령입니다.(즉 가로를 세로로)

fi = {'Features':X.columns.tolist(), 'Importance':np.transpose(lr\_clf.coef\_[0])}

importance = pd.DataFrame(fi, index=fi['Features']).sort\_values('Importance', ascending=True)

# 그래프 타이틀

title = 'Top 20 important features in predicting survival on the Titanic: Logistic Regression'

# 그래프 그리기

importance\_plotting(importance, 'Importance', 'Features', title, 20)

**# ## 5개 모델의 컬럼들의 평균 중요도로 그래프로 보기**

# 중요도가 제공되는 5가지 모델에 대한 항목 중요도 얻기

gb\_clf\_imp = pd.DataFrame({'Feature':X.columns, 'gb\_clf importance':gb\_clf.feature\_importances\_})

xgb\_clf\_imp = pd.DataFrame({'Feature':X.columns, 'xgb\_clf importance':xgb\_clf.feature\_importances\_})

rf\_clf\_imp = pd.DataFrame({'Feature':X.columns, 'rf\_clf importance':rf\_clf.feature\_importances\_})

ext\_clf\_imp = pd.DataFrame({'Feature':X.columns, 'ext\_clf importance':ext\_clf.feature\_importances\_})

ada\_clf\_imp = pd.DataFrame({'Feature':X.columns, 'ada\_clf importance':ada\_clf.feature\_importances\_})

# 이를 하나의 데이터프레임으로

importances = gb\_clf\_imp.merge(xgb\_clf\_imp, on='Feature').merge(rf\_clf\_imp, on='Feature').merge(ext\_clf\_imp, on='Feature').merge(ada\_clf\_imp, on='Feature')

# 항목당 평균 중요도

importances['Average'] = importances.mean(axis=1)

# 랭킹 정하기

importances = importances.sort\_values(by='Average', ascending=False).reset\_index(drop=True)

importances

# 중요도를 다시 데이터 프레임에 넣기

fi = {'Features':importances['Feature'], 'Importance':importances['Average']}

importance = pd.DataFrame(fi).set\_index('Features').sort\_values('Importance', ascending=True)

# 그래프 타이틀

title = 'Top 20 important features in predicting survival on the Titanic: 5 model average'

# 그래프 보기

importance\_plotting(importance, 'Importance', 'Features', title, 20)

**# ## 중요도에 따라 모델 재설정**

\* 전처리를 하고 scaling은 하지 않은 X 데이터를 집어 넣는다.

importance\_1 = importance[-381:]

importance\_1

importance\_1[371:381]

# 영양가 있는 380개만 넣기

mylist = list(importance\_1.index)

mylist

X\_1 = pd.DataFrame()

for i in mylist:

X\_1[i] = X[i]

X\_1.head()

**# 모델의 변수를 다시 정의하고**

# X\_train = train

# X\_test = test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_1, y, test\_size = 0.2, random\_state = 156, stratify=y)

# 바꿉니다.

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

**# # 다시 머신러닝 모델 만들기(위의 코드랑 같음)**

dt\_clf = DecisionTreeClassifier()

rf\_clf = RandomForestClassifier(random\_state=1)

knn\_clf = KNeighborsClassifier()

lr\_clf = LogisticRegression()

xgb\_clf = XGBClassifier()

lgbm\_clf = LGBMClassifier()

gb\_clf = GradientBoostingClassifier()

svc\_clf = SVC(probability=True)

ext\_clf = ExtraTreesClassifier()

ada\_clf = AdaBoostClassifier()

gnb\_clf = GaussianNB()

gp\_clf = GaussianProcessClassifier()

bag\_clf = BaggingClassifier()

# 리스트 준비

models = [dt\_clf, rf\_clf, knn\_clf, lr\_clf, xgb\_clf, lgbm\_clf, gb\_clf, svc\_clf, ext\_clf, ada\_clf, gnb\_clf, gp\_clf, bag\_clf]

model\_names = ['Decition Tree', 'Random Forest', 'K Nearest Neighbour', 'Logistic Regression', 'XGBoost', 'LightGBM', 'Gradient Boosting', 'SVC', 'Extra Trees', 'AdaBoost', 'Gaussian Naive Bayes', 'Gaussian Process', 'Bagging Classifier']

scores2 = {}

# 이어서 연속적으로 모델을 학습 시키고 교차 검증합니다.

for ind, mod in enumerate(models):

mod.fit(X\_train, y\_train)

acc = cross\_val\_score(mod, X\_train, y\_train, scoring = "accuracy", cv = 10)

scores2[model\_names[ind]] = acc

**# ## 결과 : 테이블로 보기**

# 결과 테이블을 만듭니다.

results = pd.DataFrame(scores2).T

results['mean'] = results.mean(1)

results

**# ## 결과 : 그래프(boxplot)으로 보기**

# 그래프로 표현

result\_df = results.sort\_values(by='mean', ascending=False)#.reset\_index()

result\_df.head(11)

result\_df = result\_df.drop(['mean'], axis=1)

sns.boxplot(data=result\_df.T, orient='h')

plt.title('Machine Learning Algorithm Accuracy Score \n')

plt.xlabel('Accuracy Score (%)');

**# # 하이퍼 파라미터 튜닝**

**# ## DecisionTree**

max\_depth = [3, None]

max\_features = [0.1, 0.2, 0.5, 0.8]

min\_samples\_split = [2, 10]

min\_samples\_leaf = [2, 10]

hyperparams = {'max\_depth': max\_depth, 'max\_features': max\_features,

'min\_samples\_split': min\_samples\_split, 'min\_samples\_leaf': min\_samples\_leaf}

gd=GridSearchCV(estimator = DecisionTreeClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## SVC**

\* Scikit-Learn에서는 3가지 모형 최적화 도구를 지원하는데 validation\_curve/ GridSearchCV/ ParameterGrid이다

\* fit 메소드를 호출하면 grid search가 자동으로 여러개의 내부 모형을 생성하고 이를 모두 실행시켜서 최적 파라미터를 찾는다.

\* bestscore는 최고 점수이고 best estimator는 최고 점수를 낸 파라미터를 가진 모형

\* c값과 gamma값은 10의 배수로 일반적으로 한다.

\* 감마 매개 변수는 단일 학습 예제의 영향이 도달하는 정도를 정의하며 낮은 값은 'far'를, 높은 값은 'close'를 나타냅니다. 감마 매개 변수는 서포트 벡터로 모델에 의해 선택된 샘플의 영향 반경의 역으로 볼 수 있습니다.

\* C 매개 변수는 의사 결정 표면의 단순성에 대한 훈련 예제의 오 분류를 제거합니다. C가 낮을수록 결정 표면이 매끄럽고 높은 C는 모델이 더 많은 샘플을 서포트 벡터로 자유롭게 선택할 수 있도록하여 모든 학습 예제를 올바르게 분류하는 것을 목표로합니다.

\* Verbose는 불리안 값으로 True로 넣으면 꼬치 꼬치 다 알려주는데, 대신 시간이 좀 더 오래 걸립니다.

\* cv =5는 5 fold로 교차 검증한다는 뜻입니다.

# 파라미터 서치

Cs = [0.01, 0.1, 1, 5, 10, 15, 20, 50]

gammas = [0.001, 0.01, 0.1]

# 파라미터 그리드 셋팅

hyperparams = {'C': Cs, 'gamma' : gammas}

# 교차검증

gd=GridSearchCV(estimator = SVC(probability=True), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

# 모델 fiting 및 결과

gd.fit(X\_train, y\_train)

display(pd.DataFrame(gd.cv\_results\_))

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Gradient Boosting Classifier**

# \* learning\_rate는 각 트리의 기여를 줄이는 역할을 합니다.

# \* n\_estimator는 각 경우의 트리 숫자입니다.

learning\_rate = [0.01, 0.05, 0.1, 0.2, 0.5]

n\_estimators = [100, 1000, 2000]

max\_depth = [3, 5, 10, 15]

hyperparams = {'learning\_rate': learning\_rate, 'n\_estimators': n\_estimators}

gd=GridSearchCV(estimator = GradientBoostingClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Logistic Regression**

\* Penalty - L1 을 사용하는 회귀 모델을 Lasso Regression이라고하고 L2를 사용하는 모델을 Ridge Regression이라고합니다. 이 둘의 주요 차이점은 페널티입니다. 릿지 회귀는 손실 함수에 페널티 항으로 계수의 "제곱 크기"를 추가합니다. L2-norm이 오차를 제곱하기 때문에 (오류> 1 인 경우 로트가 증가 함) 모델은 L1-norm보다 훨씬 큰 오차 (e vs e ^ 2)를 보게되므로 훨씬 더 민감합니다. 따라서 오류를 최소화하기 위해 모델을 조정해줍니다.

\* C는 estimator 입니다. logspace 1차원 10개 배열로 0에서 4까지를 estimator로 놓은 것입니다.

penalty = ['l1', 'l2']

C = np.logspace(0, 4, 10)

hyperparams = {'penalty': penalty, 'C': C}

gd=GridSearchCV(estimator = LogisticRegression(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## XGBoost**

# ### XGBoost Step 1.

learning\_rate = [0.001, 0.005, 0.01, 0.05, 0.1, 0.2]

n\_estimators = [10, 50, 100, 250, 500, 1000]

hyperparams = {'learning\_rate': learning\_rate, 'n\_estimators': n\_estimators}

gd=GridSearchCV(estimator = XGBClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### XGB Step 2.

max\_depth = [3, 4, 5, 6, 7, 8, 9, 10]

min\_child\_weight = [1, 2, 3, 4, 5, 6]

hyperparams = {'max\_depth': max\_depth, 'min\_child\_weight': min\_child\_weight}

gd=GridSearchCV(estimator = XGBClassifier(learning\_rate=0.2, n\_estimators=10), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### XGB Step 3.

gamma = [i\*0.1 for i in range(0,5)]

hyperparams = {'gamma': gamma}

gd=GridSearchCV(estimator = XGBClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### XGB Step 4

subsample = [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]

colsample\_bytree = [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]

hyperparams = {'subsample': subsample, 'colsample\_bytree': colsample\_bytree}

gd=GridSearchCV(estimator = XGBClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1, gamma=0), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### XGB Step 5

reg\_alpha = [1e-5, 1e-2, 0.1, 1, 100]

hyperparams = {'reg\_alpha': reg\_alpha}

gd=GridSearchCV(estimator = XGBClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1, gamma=0, subsample=1, colsample\_bytree=1),

param\_grid = hyperparams, verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## LightGBM**

# ### LightGBM Step 1.

learning\_rate = [0.001, 0.005, 0.01, 0.05, 0.1, 0.2]

n\_estimators = [10, 50, 100, 250, 500, 1000]

hyperparams = {'learning\_rate': learning\_rate, 'n\_estimators': n\_estimators}

gd=GridSearchCV(estimator = LGBMClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### LightGBM Step 2.

max\_depth = [3, 4, 5, 6, 7, 8, 9, 10]

min\_child\_weight = [1, 2, 3, 4, 5, 6]

hyperparams = {'max\_depth': max\_depth, 'min\_child\_weight': min\_child\_weight}

gd=GridSearchCV(estimator = LGBMClassifier(learning\_rate=0.2, n\_estimators=10), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### LightGBM Step 3.

gamma = [i\*0.1 for i in range(0,5)]

hyperparams = {'gamma': gamma}

gd=GridSearchCV(estimator = LGBMClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### LightGBM Step 4

subsample = [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]

colsample\_bytree = [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]

hyperparams = {'subsample': subsample, 'colsample\_bytree': colsample\_bytree}

gd=GridSearchCV(estimator = LGBMClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1, gamma=0), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

# ### LightGBM Step 5

reg\_alpha = [1e-5, 1e-2, 0.1, 1, 100]

hyperparams = {'reg\_alpha': reg\_alpha}

gd=GridSearchCV(estimator = LGBMClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1, gamma=0, subsample=1, colsample\_bytree=1),

param\_grid = hyperparams, verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Gaussian Process**

n\_restarts\_optimizer = [0, 1, 2, 3]

max\_iter\_predict = [1, 2, 5, 10, 20, 35, 50, 100]

warm\_start = [True, False]

hyperparams = {'n\_restarts\_optimizer': n\_restarts\_optimizer, 'max\_iter\_predict': max\_iter\_predict, 'warm\_start': warm\_start}

gd=GridSearchCV(estimator = GaussianProcessClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Adaboost.**

n\_estimators = [10, 100, 200, 500]

learning\_rate = [0.001, 0.01, 0.1, 0.5, 1, 1.5, 2]

hyperparams = {'n\_estimators': n\_estimators, 'learning\_rate': learning\_rate}

gd=GridSearchCV(estimator = AdaBoostClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## KNN**

n\_neighbors = [1, 2, 3, 4, 5]

algorithm = ['auto']

weights = ['uniform', 'distance']

leaf\_size = [1, 2, 3, 4, 5, 10]

hyperparams = {'algorithm': algorithm, 'weights': weights, 'leaf\_size': leaf\_size,

'n\_neighbors': n\_neighbors}

gd=GridSearchCV(estimator = KNeighborsClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

# Fitting model and return results

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Random Forest.**

n\_estimators = [10, 50, 100, 200]

max\_depth = [3, None]

max\_features = [0.1, 0.2, 0.5, 0.8]

min\_samples\_split = [2, 6]

min\_samples\_leaf = [2, 6]

hyperparams = {'n\_estimators': n\_estimators, 'max\_depth': max\_depth, 'max\_features': max\_features,

'min\_samples\_split': min\_samples\_split, 'min\_samples\_leaf': min\_samples\_leaf}

gd=GridSearchCV(estimator = RandomForestClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Extra Trees**

n\_estimators = [10, 25, 50, 75, 100]

max\_depth = [3, None]

max\_features = [0.1, 0.2, 0.5, 0.8]

min\_samples\_split = [2, 10]

min\_samples\_leaf = [2, 10]

hyperparams = {'n\_estimators': n\_estimators, 'max\_depth': max\_depth, 'max\_features': max\_features,

'min\_samples\_split': min\_samples\_split, 'min\_samples\_leaf': min\_samples\_leaf}

gd=GridSearchCV(estimator = ExtraTreesClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# ## Bagging Classifier**

n\_estimators = [10, 50, 75, 100, 200]

max\_samples = [0.1, 0.2, 0.5, 0.8, 1.0]

max\_features = [0.1, 0.2, 0.5, 0.8, 1.0]

hyperparams = {'n\_estimators': n\_estimators, 'max\_samples': max\_samples, 'max\_features': max\_features}

gd=GridSearchCV(estimator = BaggingClassifier(), param\_grid = hyperparams,

verbose=True, cv=5, scoring = "accuracy", n\_jobs=-1)

gd.fit(X\_train, y\_train)

print(gd.best\_score\_)

print(gd.best\_params\_)

**# # 다시 하이퍼파라미터 넣은 머신러닝 모델 만들기(재 트레이닝)**

# 튜닝 모델 시작

# sample을 split하는 것은 전체데이터 80%를 트레인셋에 20%는 테스트셋에 줌

dt\_clf = DecisionTreeClassifier(max\_depth=None, max\_features=0.5, min\_samples\_leaf=10, min\_samples\_split=10)

rf\_clf = RandomForestClassifier(max\_depth=None, max\_features=0.1, min\_samples\_leaf=2, min\_samples\_split=2, n\_estimators=100, random\_state=1)

knn = KNeighborsClassifier(leaf\_size=1, n\_neighbors=3, weights='distance')

log = LogisticRegression(C=59.94842503189409, penalty='l2')

xgb = XGBClassifier(learning\_rate=0.2, n\_estimators=10, max\_depth=6,

min\_child\_weight=1, gamma=0, subsample=1, colsample\_bytree=1, reg\_alpha=1e-05)

lgbm\_clf = LGBMClassifier(max\_depth=3, min\_child\_weight=6)

gbc = GradientBoostingClassifier(learning\_rate=0.2, max\_depth=3, n\_estimators=1000)

svc = SVC(probability=True, gamma=0.001, C=1)

ext = ExtraTreesClassifier(max\_depth=None, max\_features=0.1, min\_samples\_leaf=2, min\_samples\_split=10, n\_estimators=75, random\_state=1)

ada = AdaBoostClassifier(learning\_rate=1, n\_estimators=200, random\_state=1)

gpc = GaussianProcessClassifier(max\_iter\_predict=1, n\_restarts\_optimizer=0, warm\_start=True)

bag = BaggingClassifier(max\_features=0.5, max\_samples=1.0, n\_estimators=10, random\_state=1)

# 리스트

models = [dt\_clf, rf\_clf, knn\_clf, lr\_clf, xgb\_clf, lgbm\_clf, gb\_clf, svc\_clf, ext\_clf, ada\_clf, gnb\_clf, gp\_clf, bag\_clf]

model\_names = ['Decition Tree', 'Random Forest', 'K Nearest Neighbour', 'Logistic Regression', 'XGBoost', 'LightGBM', 'Gradient Boosting', 'SVC', 'Extra Trees', 'AdaBoost', 'Gaussian Naive Bayes', 'Gaussian Process', 'Bagging Classifier']

scores3 = {}

# Sequentially fit and cross validate all models

for ind, mod in enumerate(models):

mod.fit(X\_train, y\_train)

acc = cross\_val\_score(mod, X\_train, y\_train, scoring = "accuracy", cv = 10)

scores3[model\_names[ind]] = acc

**# ## 결과 : 테이블로 보기**

results = pd.DataFrame(scores3).T

results['mean'] = results.mean(1)

result\_df = results.sort\_values(by='mean', ascending=False)

result\_df

**# ## 결과 : 그래프(boxplot)으로 보기**

result\_df = result\_df.drop(['mean'], axis=1)

sns.boxplot(data=result\_df.T, orient='h')

plt.title('Machine Learning Algorithm Accuracy Score \n')

plt.xlabel('Accuracy Score (%)');

**# # 마지막 보팅(알고리즘 뭘로 쓸지 결정)**

**# ## 하드 보팅**

#튜닝한 파라미터로 하드보팅

grid\_hard = VotingClassifier(estimators = [('Decition Tree', dt\_clf),

('Random Forest', rf\_clf),

('Logistic Regression', lg\_clf),

('XGBoost', xgb\_clf),

('LightGBM', lgbm\_clf),

('Gradient Boosting', gb\_clf),

('Extra Trees', ext\_clf),

('AdaBoost', ada\_clf),

('Gaussian Process', gp\_clf),

('SVC', svc\_clf),

('K Nearest Neighbour', knn\_clf),

('Bagging Classifier', bag\_clf)], voting = 'hard')

grid\_hard\_cv = model\_selection.cross\_validate(grid\_hard, X\_train, y\_train, cv=10)

grid\_hard.fit(X\_train, y\_train)

print("Hard voting on test set score mean: {:.2f}". format(grid\_hard\_cv['test\_score'].mean() \* 100))

**# ## 소프트 보팅**

grid\_soft = VotingClassifier(estimators = [('Decition Tree', dt\_clf),

('Random Forest', rf\_clf),

('Logistic Regression', lg\_clf),

('XGBoost', xgb\_clf),

('LightGBM', lgbm\_clf),

('Gradient Boosting', gb\_clf),

('Extra Trees', ext\_clf),

('AdaBoost', ada\_clf),

('Gaussian Process', gp\_clf),

('SVC', svc\_clf),

('K Nearest Neighbour', knn\_clf),

('Bagging Classifier', bag\_clf)], voting = 'soft')

grid\_soft\_cv = model\_selection.cross\_validate(grid\_soft, X\_train, y\_train, cv=10)

grid\_soft.fit(X\_train, y\_train)

print("Soft voting on test set score mean: {:.2f}". format(grid\_soft\_cv['test\_score'].mean() \* 100))

**# # 마지막 모델 예측**

**# ## 하드보팅 예측**

# Final predictions2

predictions = grid\_hard.predict(X\_test)

submission = pd.concat([pd.DataFrame(passId), pd.DataFrame(predictions)], axis = 'columns')

submission.columns = ["PassengerId", "Survived"]

submission.to\_csv('titanic\_submission1.csv', header = True, index = False)

**# ## 소프트보팅 예측**

# Final predictions

predictions = grid\_soft.predict(X\_test)

submission = pd.concat([pd.DataFrame(passId), pd.DataFrame(predictions)], axis = 'columns')

submission.columns = ["PassengerId", "Survived"]

submission.to\_csv('titanic\_submission2.csv', header = True, index = False)

**# # 평가**

\* 정확도(Accuracy) = (예측 결과가 동일한 데이터 건수) / (전체 예측 데이터 건수)

\* 정밀도란 모델이 True라고 분류한 것 중에서 실제 True인 것의 비율입니다. 즉, 아래와 같은 식으로 표현할 수 있습니다.

\* 재현율이란 실제 True인 것 중에서 모델이 True라고 예측한 것의 비율입니다.

\* 암 검사를 위해 병원을 찾아온 사람에게 암여부를 예측할때에는 재현율이 중요하다, 즉 실제로 암인데 암이 아니라고 예측하면 큰일나기 때문이다.

\* 메일이 왔는데 스팸메일여부를 판단하는 것은 정밀도가 중요하다.

\* 이렇듯 어떤것에 사용하느냐에 따라 달라지며 대부분은 재현율이 더 중요해 많이 사용하고 있다.

\*F1 score는 Precision과 Recall의 조화평균입니다.

\* ROC-curve은 curve가 왼쪽 위 모서리에 가까울수록 모델의 성능이 좋다고 평가합니다. 즉, Recall이 크고 Fall-out이 작은 모형이 좋은 모형인 것입니다.

\* 오차행렬

예측 클래스

실제 클래스 [TN FN]

[FN TP]

**# ## 평가 함수 모음**

def total\_get\_clf\_eval(model, X\_train, X\_test, y\_train, y\_test, pred=None, pred\_proba\_c1=None):

# 알고리즘

model.fit(X\_train, y\_train)

pred = model.predict(X\_test)

pred\_proba\_c1 = model.predict\_proba(X\_test)[:, 1]

# 평가 항목

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\_c1)

print('오차 행렬')

print(confusion)

# ROC-AUC print 추가

print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1: {3:.4f}, AUC:{4:.4f}'.format(accuracy, precision, recall, f1, roc\_auc))

# precision\_recall\_curve\_plot

precision\_recall\_curve\_plot(y\_test, pred\_proba\_c1)

# roc\_curve\_plot

roc\_curve\_plot(y\_test , pred\_proba\_c1)

def precision\_recall\_curve\_plot(y\_test=None, pred\_proba\_c1=None):

# threshold ndarray와 이 threshold에 따른 정밀도, 재현율 ndarray 추출.

precisions, recalls, thresholds = precision\_recall\_curve( y\_test, pred\_proba\_c1)

# X축을 threshold값으로, Y축은 정밀도, 재현율 값으로 각각 Plot 수행. 정밀도는 점선으로 표시

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')

# threshold 값 X 축의 Scale을 0.1 단위로 변경

start, end = plt.xlim()

plt.xticks(np.round(np.arange(start, end, 0.1),2))

# x축, y축 label과 legend, 그리고 grid 설정

plt.xlabel('Threshold value'); plt.ylabel('Precision and Recall value')

plt.legend(); plt.grid()

plt.show()

def roc\_curve\_plot(y\_test , pred\_proba\_c1):

# 임곗값에 따른 FPR, TPR 값을 반환 받음.

fprs , tprs , thresholds = roc\_curve(y\_test ,pred\_proba\_c1)

# ROC Curve를 plot 곡선으로 그림.

plt.plot(fprs , tprs, label='ROC')

# 가운데 대각선 직선을 그림.

plt.plot([0, 1], [0, 1], 'k--', label='Random')

# FPR X 축의 Scale을 0.1 단위로 변경, X,Y 축명 설정등

start, end = plt.xlim()

plt.xticks(np.round(np.arange(start, end, 0.1),2))

plt.xlim(0,1); plt.ylim(0,1)

plt.xlabel('FPR( 1 - Sensitivity )'); plt.ylabel('TPR( Recall )')

plt.legend()

plt.show()

**# ## 전체보기**

for i, model in enumerate(models):

print(model\_names[i],'\n')

total\_get\_clf\_eval(model, X\_train, X\_test, y\_train, y\_test, pred=None, pred\_proba\_c1=None)

print('-------------------------------------------------------------------------------------------------\n')

**# ## 모델별 보기**

# ### 수치 평가

# 수정된 get\_clf\_eval() 함수

def get\_clf\_eval(model, X\_train, X\_test, y\_train, y\_test, pred=None, pred\_proba\_c1=None):

# 알고리즘

model.fit(X\_train, y\_train)

pred = model.predict(X\_test)

pred\_proba\_c1 = model.predict\_proba(X\_test)[:, 1]

# 평가 항목

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\_c1)

print('오차 행렬')

print(confusion)

# ROC-AUC print 추가

print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1: {3:.4f}, AUC:{4:.4f}'.format(accuracy, precision, recall, f1, roc\_auc))

get\_clf\_eval(rf\_clf, X\_train, X\_test, y\_train, y\_test, pred=None, pred\_proba\_c1=None)

# ### 임계값 조정 후 수치 평가

# 알고리즘 모델별로 임계값을 조정하면서 하나하나 봐야 함

from sklearn.preprocessing import Binarizer

# model = 모델약어, pred\_proba\_c1 = pred\_proba로 예측 후 1번쨰 컬럼, thresholds =

def get\_eval\_by\_threshold(model, X\_train, X\_test, y\_train, y\_test, pred\_proba\_c1, thresholds):

# thresholds 리스트 객체내의 값을 차례로 iteration하면서 Evaluation 수행.

for custom\_threshold in thresholds:

binarizer = Binarizer(threshold=custom\_threshold).fit(pred\_proba\_c1)

custom\_predict = binarizer.transform(pred\_proba\_c1)

print('임계값:',custom\_threshold,'\n')

# roc\_auc\_score 관련 수정

get\_clf\_eval(model, X\_train, X\_test, y\_train, y\_test, custom\_predict, pred\_proba\_c1)

print('-------------------------------------------------------------------------------------------------\n')

# 임계값을 변경해서 넣어야 함

thresholds = [0.3 , 0.33 ,0.36,0.39, 0.42 , 0.45 ,0.48, 0.50]

# 모델을 변경해서 넣어야 함

pred\_proba\_c1 = rf\_clf.predict\_proba(X\_test)[:,1]

get\_eval\_by\_threshold(rf\_clf, X\_train, X\_test, y\_train, y\_test, pred\_proba\_c1.reshape(-1,1), thresholds)

# ### precision\_recall\_curve 그래프

def precision\_recall\_curve\_plot(y\_test=None, pred\_proba\_c1=None):

# threshold ndarray와 이 threshold에 따른 정밀도, 재현율 ndarray 추출.

precisions, recalls, thresholds = precision\_recall\_curve(y\_test, pred\_proba\_c1)

# X축을 threshold값으로, Y축은 정밀도, 재현율 값으로 각각 Plot 수행. 정밀도는 점선으로 표시

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')

# threshold 값 X 축의 Scale을 0.1 단위로 변경

start, end = plt.xlim()

plt.xticks(np.round(np.arange(start, end, 0.1),2))

# x축, y축 label과 legend, 그리고 grid 설정

plt.xlabel('Threshold value'); plt.ylabel('Precision and Recall value')

plt.legend(); plt.grid()

plt.show()

pred\_proba\_c1 = rf\_clf.predict\_proba(X\_test)[:,1]

precision\_recall\_curve\_plot(y\_test, pred\_proba\_c1)

# ### roc\_curve 그래프

def roc\_curve\_plot(y\_test , pred\_proba\_c1):

# 임곗값에 따른 FPR, TPR 값을 반환 받음.

fprs , tprs , thresholds = roc\_curve(y\_test ,pred\_proba\_c1)

# ROC Curve를 plot 곡선으로 그림.

plt.plot(fprs , tprs, label='ROC')

# 가운데 대각선 직선을 그림.

plt.plot([0, 1], [0, 1], 'k--', label='Random')

# FPR X 축의 Scale을 0.1 단위로 변경, X,Y 축명 설정등

start, end = plt.xlim()

plt.xticks(np.round(np.arange(start, end, 0.1),2))

plt.xlim(0,1); plt.ylim(0,1)

plt.xlabel('FPR( 1 - Sensitivity )'); plt.ylabel('TPR( Recall )')

plt.legend()

plt.show()

pred\_proba\_c1 = rf\_clf.predict\_proba(X\_test)[:,1]

roc\_curve\_plot(y\_test, pred\_proba\_c1)