import os

import IPython

import multiprocessing

import copy

import pickle

import warnings

from datetime import datetime, timedelta

from time import time, sleep, mktime

from matplotlib import font\_manager as fm, rc, rcParams

import matplotlib.pyplot as plt

import seaborn as sns

from tqdm import tqdm

import re

import numpy as np

from numpy import array, nan, random as rnd, where as which

import pandas as pd

from pandas import DataFrame as dataframe, Series as series, isna, read\_csv

from pandas.tseries.offsets import DateOffset

import statsmodels.api as sm

from sklearn import preprocessing as prep

from sklearn.impute import KNNImputer

from sklearn.model\_selection import train\_test\_split as tts, GridSearchCV as GridTuner, StratifiedKFold, KFold

from sklearn.feature\_selection import SelectFromModel

from sklearn.preprocessing import OneHotEncoder, StandardScaler, MinMaxScaler, RobustScaler

from sklearn import metrics

from sklearn.pipeline import make\_pipeline

from sklearn import linear\_model as lm

from sklearn.discriminant\_analysis import QuadraticDiscriminantAnalysis as qda

from sklearn import svm

import lightgbm as lgb

import xgboost as xgb

import catboost as cat

from sklearn import neighbors as knn

from sklearn import ensemble

# # ===== tensorflow =====

# import tensorflow as tf

# from tensorflow.keras.models import Model

# from tensorflow.keras import layers

# from tensorflow.keras import activations

# from tensorflow.keras import optimizers

# from tensorflow.keras import metrics as tf\_metrics

# from tensorflow.keras import callbacks as tf\_callbacks

# from tqdm.keras import TqdmCallback

# from scikeras.wrappers import KerasClassifier, KerasRegressor

# import tensorflow\_addons as tfa

# import keras\_tuner as kt

# from keras\_tuner import HyperModel

# ===== NLP =====

from selenium import webdriver

from konlpy.tag import Okt

from KnuSentiLex.knusl import KnuSL

# ===== task specific =====

import pykrx

# display setting

warnings.filterwarnings(action='ignore')

rcParams['axes.unicode\_minus'] = False

pd.set\_option('display.max\_columns', 100)

pd.set\_option('display.max\_rows', 100)

pd.set\_option('display.width', 1000)

# # font setting

# font\_path = 'myfonts/NanumSquareB.ttf'

# font\_obj = fm.FontProperties(fname=font\_path, size=12).get\_name()

# rc('font', family=font\_obj)

# %reset -f

# ===== utility functions =====

# label encoding for categorical column with excepting na value

class MyLabelEncoder:

def \_\_init\_\_(self, preset={}):

# dic\_cat format -> {"col\_name": {"value": replace}}

self.dic\_cat = preset

def fit\_transform(self, data\_x, col\_names):

tmp\_x = copy.deepcopy(data\_x)

for i in col\_names:

# type check

if not ((tmp\_x[i].dtype.name == "object") or (tmp\_x[i].dtype.name == "category")):

print(F"WARNING : {i} is not object or category")

# if key is not in dic, update dic

if i not in self.dic\_cat.keys():

tmp\_dic = dict.fromkeys(sorted(set(tmp\_x[i]).difference([nan])))

label\_cnt = 0

for j in tmp\_dic.keys():

tmp\_dic[j] = label\_cnt

label\_cnt += 1

self.dic\_cat[i] = tmp\_dic

# transform value which is not in dic to nan

tmp\_x[i] = tmp\_x[i].astype("object")

conv = tmp\_x[i].replace(self.dic\_cat[i])

for conv\_idx, j in enumerate(conv):

if j not in self.dic\_cat[i].values():

conv[conv\_idx] = nan

# final return

tmp\_x[i] = conv.astype("float")

return tmp\_x

def transform(self, data\_x):

tmp\_x = copy.deepcopy(data\_x)

for i in list(self.dic\_cat.keys()):

if not ((tmp\_x[i].dtype.name == "object") or (tmp\_x[i].dtype.name == "category")):

print(F"WARNING : {i} is not object or category")

# transform value which is not in dic to nan

tmp\_x[i] = tmp\_x[i].astype("object")

conv = tmp\_x[i].replace(self.dic\_cat[i])

for conv\_idx, j in enumerate(conv):

if j not in self.dic\_cat[i].values():

conv[conv\_idx] = nan

# final return

tmp\_x[i] = conv.astype("float")

return tmp\_x

def clear(self):

self.dic\_cat = {}

class MyOneHotEncoder:

def \_\_init\_\_(self, label\_preset={}):

self.dic\_cat = {}

self.label\_preset = label\_preset

def fit\_transform(self, data\_x, col\_names):

tmp\_x = dataframe()

for i in data\_x:

if i not in col\_names:

tmp\_x = pd.concat([tmp\_x, dataframe(data\_x[i])], axis=1)

else:

if not ((data\_x[i].dtype.name == "object") or (data\_x[i].dtype.name == "category")):

print(F"WARNING : {i} is not object or category")

self.dic\_cat[i] = OneHotEncoder(sparse=False, handle\_unknown="ignore")

conv = self.dic\_cat[i].fit\_transform(dataframe(data\_x[i])).astype("int")

col\_list = []

for j in self.dic\_cat[i].categories\_[0]:

if i in self.label\_preset.keys():

for k, v in self.label\_preset[i].items():

if v == j:

col\_list.append(str(i) + "\_" + str(k))

else:

col\_list.append(str(i) + "\_" + str(j))

conv = dataframe(conv, columns=col\_list)

tmp\_x = pd.concat([tmp\_x, conv], axis=1)

return tmp\_x

def transform(self, data\_x):

tmp\_x = dataframe()

for i in data\_x:

if not i in list(self.dic\_cat.keys()):

tmp\_x = pd.concat([tmp\_x, dataframe(data\_x[i])], axis=1)

else:

if not ((data\_x[i].dtype.name == "object") or (data\_x[i].dtype.name == "category")):

print(F"WARNING : {i} is not object or category")

conv = self.dic\_cat[i].transform(dataframe(data\_x[i])).astype("int")

col\_list = []

for j in self.dic\_cat[i].categories\_[0]:

if i in self.label\_preset.keys():

for k, v in self.label\_preset[i].items():

if v == j: col\_list.append(str(i) + "\_" + str(k))

else:

col\_list.append(str(i) + "\_" + str(j))

conv = dataframe(conv, columns=col\_list)

tmp\_x = pd.concat([tmp\_x, conv], axis=1)

return tmp\_x

def clear(self):

self.dic\_cat = {}

self.label\_preset = {}

class MyKNNImputer:

def \_\_init\_\_(self, k=5):

self.imputer = KNNImputer(n\_neighbors=k)

self.cat\_dic = {}

def fit\_transform(self, x, y, cat\_vars=None):

naIdx = dict.fromkeys(cat\_vars)

for i in cat\_vars:

self.cat\_dic[i] = diff(list(sorted(set(x[i]))), [nan])

naIdx[i] = list(which(array(x[i].isna()))[0])

x\_imp = dataframe(self.imputer.fit\_transform(x, y), columns=x.columns)

# if imputed categorical value are not in the range, adjust the value

for i in cat\_vars:

x\_imp[i] = x\_imp[i].apply(lambda x: int(round(x, 0)))

for j in naIdx[i]:

if x\_imp[i][j] not in self.cat\_dic[i]:

if x\_imp[i][j] < self.cat\_dic[i][0]:

x\_imp[i][naIdx[i]] = self.cat\_dic[i][0]

elif x\_imp[i][j] > self.cat\_dic[i][0]:

x\_imp[i][naIdx[i]] = self.cat\_dic[i][len(self.cat\_dic[i]) - 1]

return x\_imp

def transform(self, x):

naIdx = dict.fromkeys(self.cat\_vars)

for i in self.cat\_dic.keys():

naIdx[i] = list(which(array(x[i].isna()))[0])

x\_imp = dataframe(self.imputer.transform(x), columns=x.columns)

# if imputed categorical value are not in the range, adjust the value

for i in self.cat\_dic.keys():

x\_imp[i] = x\_imp[i].apply(lambda x: int(round(x, 0)))

for j in naIdx[i]:

if x\_imp[i][j] not in self.cat\_dic[i]:

if x\_imp[i][j] < self.cat\_dic[i][0]:

x\_imp[i][naIdx[i]] = self.cat\_dic[i][0]

elif x\_imp[i][j] > self.cat\_dic[i][0]:

x\_imp[i][naIdx[i]] = self.cat\_dic[i][len(self.cat\_dic[i]) - 1]

return x\_imp

def clear(self):

self.imputer = None

self.cat\_dic = {}

def easyIO(x=None, path=None, op="r"):

tmp = None

if op == "r":

with open(path, "rb") as f:

tmp = pickle.load(f)

return tmp

elif op == "w":

tmp = {}

print(x)

if type(x) is dict:

for k in x.keys():

if "MLP" in k:

tmp[k] = {}

for model\_comps in x[k].keys():

if model\_comps != "model":

tmp[k][model\_comps] = copy.deepcopy(x[k][model\_comps])

print(F"INFO : {k} model is removed (keras)")

else:

tmp[k] = x[k]

if input("Write [y / n]: ") == "y":

with open(path, "wb") as f:

pickle.dump(tmp, f)

print("operation success")

else:

print("operation fail")

else:

print("Unknown operation type")

def diff(first, second):

second = set(second)

return [item for item in first if item not in second]

def findIdx(data\_x, col\_names):

return [int(i) for i, j in enumerate(data\_x) if j in col\_names]

def orderElems(for\_order, using\_ref):

return [i for i in using\_ref if i in for\_order]

# concatenate by row

def ccb(df1, df2):

if type(df1) == series:

tmp\_concat = series(pd.concat([dataframe(df1), dataframe(df2)], axis=0, ignore\_index=True).iloc[:,0])

tmp\_concat.reset\_index(drop=True, inplace=True)

elif type(df1) == dataframe:

tmp\_concat = pd.concat([df1, df2], axis=0, ignore\_index=True)

tmp\_concat.reset\_index(drop=True, inplace=True)

elif type(df1) == np.ndarray:

tmp\_concat = np.concatenate([df1, df2], axis=0)

else:

print("Unknown Type: return 1st argument")

tmp\_concat = df1

return tmp\_concat

def change\_width(ax, new\_value):

for patch in ax.patches :

current\_width = patch.get\_width()

adj\_value = current\_width - new\_value

# we change the bar width

patch.set\_width(new\_value)

# we recenter the bar

patch.set\_x(patch.get\_x() + adj\_value \* .5)

def week\_of\_month(date):

month = date.month

week = 0

while date.month == month:

week += 1

date -= timedelta(days=7)

return week

def dispPerformance(result\_dic, result\_metrics):

perf\_table = dataframe(columns=result\_metrics)

for k, v in result\_dic.items():

perf\_table = pd.concat([perf\_table, v["performance"]], ignore\_index=True, axis=0)

print(perf\_table)

return perf\_table

from pykrx import stock