```
# Hello World project
# python3

# پیش بینی ; جهت قیمت در پایتون
# برای شروع کدنویسی، کتابخانه‌های مورد نیاز را فراخوانی می‌کنیم:

import numpy as np
import yfinance as yf
import sklearn.dummy as dm
import sklearn.metrics as met
import matplotlib.pyplot as plt
import sklearn.linear_model as lm
import sklearn.preprocessing as pp

# حال تنظیمات مورد نیاز را اعمال می‌کنیم:

np.random.seed(0)
plt.style.use('ggplot')

# حال مجموعه داده مربوط به کل تاریخچه روزانه نماد ETH/USD را دریافت می‌کنیم:

Ticker = 'ETH-USD'
Interval = '1d'
Period = 'max'

DF = yf.download(tickers=Ticker, interval=Interval,
period=Period)

# اکنون مجموعه داده را بررسی می‌کنیم که تا از صحت آن مطمئن شویم:
```

```
print(DF.head())
print(DF.tail())

# حال می‌توانیم درصد تغییرات نسبی در هر روز را محاسبه کنیم:

DF['RPC'] = 100 * (DF['Close'] / DF['Open'] - 1)

# حال می‌توانیم یک نمودار هیستوگرام (Histogram Plot) برای این متغیر رسم کنیم:

plt.hist(DF['RPC'], bins=41, color='b', alpha=0.6)
plt.title('ETH-USD Relative Percentage Change')
plt.xlabel('Relative Change (%)')
plt.ylabel('Frequency')
plt.show()

# حال می‌توانیم با استفاده از روش دامنه میان چارکی (Interquartile Rage) مقادیر پرت (Outlier) را اصلاح کنیم:

k = 1.5
q1 = DF['RPC'].quantile(0.25)
q3 = DF['RPC'].quantile(0.75)
iqr = q3 - q1
lb = q1 - k * iqr
ub = q3 + k * iqr
DF['RPC'] = DF['RPC'].clip(lower=lb, upper=ub)

# حال می‌توانیم داده‌های ستون RPC را به صورت آرایه دریافت کنیم:
```

```
S = DF['RPC'].to_numpy()

# اکنون تابع Lag را وارد برنامه می‌کنیم:

def Lag(S:np.ndarray, L:int):
    nD0 = S.size
    nD = nD0 - L
    X = np.zeros((nD, L))
    Y = np.zeros((nD, 1))
    for i in range(nD):
        X[i, :] = S[i:i + L]
        Y[i, 0] = S[i + L]
    return X, Y

    # و برای استفاده از آن، به شکل زیر عمل می‌کنیم:

nLag = 30
X0, Y0 = Lag(S, nLag)

#  حال داده‌ها را به دو مجموعه داده آموزش (Train Dataset)
# و آزمایش (Test Dataset) تقسیم می‌کنیم:

sTrain = 0.8
nDtr = int(sTrain * X0.shape[0])

trX0 = X0[:nDtr]
teX0 = X0[nDtr:]
trY0 = Y0[:nDtr]
teY0 = Y0[nDtr:]
```

```
#درصد تغییرات نسبی، مقیاس مناسبی ارائه نمی‌دهد
#  مقیاس آن‌ها را به شکل زیر اصلاح می‌کنیم:

SSX = pp.StandardScaler()
trX = SSX.fit_transform(trX0)
teX = SSX.transform(teX0)

#  تعریف (Discretizer) برای مقادیر ویژگی هدف، باید یک تابع گسسته‌ساز کنیم
#  و مقادیر را در کلاس مربوط به خود قرار دهیم. برای این کار معمولاً ۳ دسته در نظر می‌گیرم:
#  تعریف می‌کنیم: (Threshold) برای این کار، یک مقدار مرزی
#  اگر تغییرات کمتر از قرینه Threshold بود، کاهش قیمت رخ داده است. (دسته ۰)
#  اگر تغییرات کمتر از قرینه Threshold بود، کاهش قیمت رخ داده است. (دسته ۰)
#  اگر تغییرات بیشتر از Threshold بود، افزایش قیمت رخ داده است. (دسته ۲)
#  در غیر این صورت، تغییرات خنثی بوده است. (دسته ۱)

#  برای این کار، یک تابع Discretizer تعریف می‌کنیم که در ورودی ماتریس Y0 و مقدار Threshold را دریافت می‌کند:

def Discretizer(Y0:np.ndarray, TH:float):

    #ابتدا اندازه داده را محاسبه می‌کنیم:

    def Discretizer(Y0:np.ndarray, TH:float):
        nD = Y0.size
```

حال یک ماتریس خالی برای ذخیره دسته هر داده ایجاد می‌کنیم:

```
def Discretizer(Y0:np.ndarray, TH:float):
    nD = Y0.size
    Y = np.zeros((nD, 1))
```

حال می‌توانیم یک حلقه ایجاد کرده و برای هر داده، شرطهای گفته شده را بررسی کنیم:

```
def Discretizer(Y0:np.ndarray,
TH:float):
    nD = Y0.size
    Y = np.ones((nD, 1))
    for i in range(nD):
        if Y0[i] < -TH:
           Y[i, 0] = 0
        elif Y0[i] > +TH:
            Y[i, 0] = 2

            return Y
```

به این ترتیب، تابع مورد نظر پیاده‌سازی شد. حال برای استفاده از تابع، به شکل زیر می‌نویسیم:

```
TH = 2
trY = Discretizer(trY0, TH)
teY = Discretizer(teY0, TH)
```

توجه داشته باشید که در خروجی کد فوق، تغییرات بین ۲- و ۲+ به عنوان حرکات

خنثی در نظر گرفته می‌شود
تنظیم مقدار TH بسیار حائز اهمیت است.
حال برای استفاده از ماتریس Y برای آموزش و آزمایش مدل، آن‌ها را به شکل تک‌بُعدی تغییر می‌دهیم:

```
trY = trY.reshape(-1)
teY = teY.reshape(-1)
```

اکنون می‌توانیم مدل رگرسیون لجستیک را ایجاد کرده و آموزش دهیم:

```
Model = lm.LogisticRegression()
Model.fit(trX, trY)
```

حال می‌توانیم برای داده‌های آموزش و آزمایش پیش‌بینی‌های مدل را دریافت کنیم:

```
trPr = Model.predict(trX)
tePr = Model.predict(teX)
```

اکنون می‌توانیم گزارش طبقه‌بندی را به شکل زیر محاسبه و نمایش دهیم:

```
trCR = met.classification_report(trY, trPr)
teCR = met.classification_report(teY, tePr)

print(f'Train Classification Report:\n{trCR}')
print('_'*60)
print(f'Test  Classification Report:\n{teCR}')
```

برای F1 Score Macro Average به این ترتیب، مشاهده می‌کنیم که
داده‌های آموزش ۰٫۳۶
و برای داده‌های آزمایش ۰٫۲۷ است که نتایج نه‌چندان مطلوبی است.

برای رفع این مشکل، وزن هر دسته را به شکل زیر محاسبه می‌کنیم:

```
nClass = 3
nTotal = trY.size
Ns = {i: trY[trY == i].size for i in range(nClass)}
W = {i: (nTotal - Ns[i])/((nClass - 1) * nTotal)
for i in range(nClass)}
```

حال می‌توانیم دیکشنری وزن را به شکل زیر در تعریف مدل استفاده کنیم:

```
Model = lm.LogisticRegression(class_weight=W)
```

برای F1 Score Macro Average به این ترتیب، مشاهده می‌کنیم که داده‌های آموزش و آزمایش
ه ترتیب برابر ۰,۴۲ و ۰,۲۹ می‌شود. به این ترتیب، مشاهده می‌کنیم
که ۰,۰۶ واحد در مجموعه داده آموزش و ۰,۰۲ واحد در مجموعه داده آزمایش بهبود رخ داده است.
آموزش داد و Dummy Classifier برای درک بهتر این دقت‌ها، می‌توان یک نتایج آن را با مدلِ آموزش‌دیده مقایسه کرد:

```
Dummy =
dm.DummyClassifier(strategy='most_frequent')
Dummy.fit(trX, trY)

trPr = Dummy.predict(trX)
tePr = Dummy.predict(teX)

trF1ScoreMA = met.f1_score(trY, trPr,
average='macro')
teF1ScoreMA = met.f1_score(teY, tePr,
```

```
average='macro')

print(f'Dummy Train F1 Score Macro Average:
{trF1ScoreMA}')
print(f'Dummy Test  F1 Score Macro Average:
{teF1ScoreMA}')

# برای دست یافتن به دقت‌های بالاتر، می‌توان مدل‌هایی دیگر را مقایسه کرد
# K-نزدیک‌ترین همسایه (K-Nearest Neighbors – KNN)
# جنگل تصادفی (Random Forest – RF)
# پرسپترون چند لایه (Multi-Layer Perceptron – MLP)
# ماشین بردار پشتیبان (Support Vector Machine – SVM)
```