```
%pip install -q yfinance FinMind
```

```
import IPython
print(f'IPython=={IPython.__version__}')

import numpy as np
print(f'numpy=={np.__version__}')
import pandas as pd
print(f'pandas=={pd.__version__}')

import scipy
from scipy import signal
from scipy.linalg import solve
print(f'scipy=={scipy.__version__}')
import statsmodels
import statsmodels.api as sm
from statsmodels.tsa.filters.hp_filter import hpfilter
print(f'statsmodels=={statsmodels.__version__}')
import sklearn as sk
from sklearn.metrics import mean_squared_error
print(f'sklearn=={sk.__version__}')

import yfinance as yf
print(f'yfinance=={yf.__version__}')

import FinMind
from FinMind.data import DataLoader
print(f'FinMind=={FinMind.__version__}')

import matplotlib
import matplotlib.pyplot as plt
print(f'matplotlib=={matplotlib.__version__}')
import seaborn as sns
print(f'seaborn=={sns.__version__}')

import datetime
import os
```

```
IPython==7.34.0
numpy==1.26.4
pandas==2.2.2
scipy==1.13.1
statsmodels==0.14.4
sklearn==1.5.2
yfinance==0.2.48
FinMind==1.7.3
matplotlib==3.8.0
seaborn==0.13.2
```

HPF 法

HPF假設一時間序列 S_t 可分解為，長期趨勢 S_t^* 以及短期波動 V_t
$S_t = S_t^* + V_t,\ t = 1, 2, 3, \cdots T, \text{where}$

$$S_t^* = \min\left\{\Sigma_{t=1}^{T} {V_t}^2 + \lambda\Sigma_{t=2}^{T-1}\left[(S_{t+1}^* - S_t^*) - (S_t^* - S_{t-1}^*)\right]\right\}$$

短期波動 $V_t = y_t - S_t^*$ 因此，

$$S_t^* = \min\left\{\Sigma_{t=1}^{T}(y_t - S_t^*)^2 + \lambda\Sigma_{t=2}^{T-1}\left[(S_{t+1}^* - S_t^*) - (S_t^* - S_{t-1}^*)\right]\right\}$$

給定一組 y：

```
# y = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
# T = len(y)

np.random.seed(42)
T = 100
# 生成二階整合的時間序列，加上隨機噪音
# 二階整合過程：Δ^2 y_t = noise
y = np.zeros(T)
noise = np.random.normal(0, 1, T)

for t in range(2, T):
    y[t] = 2 * y[t-1] - y[t-2] + noise[t]  # 二階整合過程的迭代公式
print(y)
```

```
[ 0.00000000e+00  0.00000000e+00  6.47688538e-01  2.81840693e+00
  4.75497195e+00  6.45740002e+00  9.73904089e+00  1.37881165e+01
  1.73677177e+01  2.14898790e+01  2.51486226e+01  2.83416364e+01
  3.17766125e+01  3.32983083e+01  3.30950863e+01  3.23295768e+01
  3.05512362e+01  2.90871429e+01  2.67150255e+01  2.29306044e+01
  2.06118321e+01  1.80672835e+01  1.55902631e+01  1.16884945e+01
  7.24234317e+00  2.90711444e+00 -2.57910787e+00 -7.68963215e+00
 -1.34007951e+01 -1.94036519e+01 -2.60082152e+01 -3.07605004e+01
 -3.55262827e+01 -4.13497760e+01 -4.63507244e+01 -5.25725165e+01
 -5.85854449e+01 -6.65580435e+01 -7.58588281e+01 -8.49627515e+01
 -9.33282083e+01 -1.01522297e+02 -1.09832034e+02 -1.18442874e+02
 -1.28532237e+02 -1.39341443e+02 -1.50611289e+02 -1.60824012e+02
 -1.70693117e+02 -1.82325262e+02 -1.93633323e+02 -2.05326467e+02
 -2.17696532e+02 -2.29454921e+02 -2.40182311e+02 -2.49978420e+02
 -2.60613747e+02 -2.71558287e+02 -2.82171563e+02 -2.91809294e+02
 -3.01926199e+02 -3.12228763e+02 -3.23637662e+02 -3.36242767e+02
 -3.48035347e+02 -3.58471687e+02 -3.68980037e+02 -3.78484854e+02
 -3.87628035e+02 -3.97416336e+02 -4.06843241e+02 -4.14732109e+02
 -4.22656804e+02 -4.29016855e+02 -4.37996651e+02 -4.46154545e+02
 -4.54225391e+02 -4.62595245e+02 -4.70873338e+02 -4.81139000e+02
 -4.91624334e+02 -5.01752555e+02 -5.10402882e+02 -5.19571480e+02
 -5.29548571e+02 -5.40027419e+02 -5.49590865e+02 -5.58825560e+02
 -5.68590015e+02 -5.77841203e+02 -5.86995313e+02 -5.95180778e+02
 -6.04068296e+02 -6.13283477e+02 -6.22890765e+02 -6.33961569e+02
 -6.44736252e+02 -6.55249880e+02 -6.65758394e+02 -6.76501496e+02]
```

在 Hodrick-Prescott 濾波器的實作中，單位矩陣 I 用來幫助建立最小化目標的函數，並與懲罰矩陣 penalty 結合，以構建平滑趨勢估計函數。

$$(I + \text{penalty}) \cdot S^* = y$$

```
I = np.eye(T)
print(I)
```

```
[[1. 0. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 0. 1.]]
```

生成一個大小為 $(T-2) \times T$ 的零矩陣 D，用來計算趨勢 S^* 的二階差分。這個矩陣最終會被填充，用來計算各時間點趨勢之間的平滑性。

在 Hodrick-Prescott 濾波器中，D 矩陣的用途是幫助衡量趨勢成分之間的變動幅度。以下是流程：

1. 建立二階差分矩陣 D 矩陣被設計為二階差分矩陣，用來計算相鄰三個時間點的趨勢變化，具體計算公式是：

$$\Delta^2 S^* = (S^*_{t+1} - S^*_t) - (S^*_t - S^*_{t-1}) = S^*_{t+1} - 2S^*_t + S^*_{t-1}$$

   這個公式用來評估趨勢在相鄰點之間的曲率，從而在最小化目標函數時對趨勢的平滑性進行懲罰。當這個差分的值較大時，說明趨勢變化劇烈，不夠平滑；反之，當差分較小時，趨勢更為平滑。

2. 初始化零矩陣

```
D = np.zeros((T-2, T))
print(D)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

3. 矩陣填充以實現二階差分

$$S^*_{t+1} - 2S^*_t + S^*_{t-1}$$

```
for i in range(T-2):
    D[i, i] = 1
    D[i, i+1] = -2
```

```
    D[i, i+2] = 1
print(D)
```

```
[[ 1. -2.  1. ...  0.  0.  0.]
 [ 0.  1. -2. ...  0.  0.  0.]
 [ 0.  0.  1. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  1.  0.  0.]
 [ 0.  0.  0. ... -2.  1.  0.]
 [ 0.  0.  0. ...  1. -2.  1.]]
```

`lamb`： 這是平滑參數，用來調整懲罰項的強度。對於季度數據，通常設置 $\lambda = 1600$，這個值決定了平滑程度。較大的 λ 會使趨勢 S^* 更加平滑，較小的 λ 則讓趨勢更加貼近原始數據。

`D.T`： D 的轉置矩陣，將 $(T-2) \times T$ 的矩陣轉換為 $T \times (T-2)$矩陣，目的是生成一個平滑矩陣，使得我們能夠對所有趨勢點的平滑程度進行調整。

`@ D`： `@` 表示矩陣乘法。通過 `D.T @ D`，我們得到了平滑的二階差分矩陣，這一矩陣可以直接用於懲罰項中，使得優化算法會偏向生成更平滑的趨勢。

```
lamb = 1600
penalty = lamb * D.T @ D
print(penalty)
```

```
[[ 1600. -3200.  1600. ...     0.     0.     0.]
 [-3200.  8000. -6400. ...     0.     0.     0.]
 [ 1600. -6400.  9600. ...     0.     0.     0.]
 ...
 [    0.     0.     0. ...  9600. -6400.  1600.]
 [    0.     0.     0. ... -6400.  8000. -3200.]
 [    0.     0.     0. ...  1600. -3200.  1600.]]
```

在 HP 濾波器中，我們希望找到一個趨勢成分 S^*，使得趨勢既符合原始數據 y 又平滑。因此，我們構建了以下優化問題來達到這個目的

$$S_t^* = \min\left\{\Sigma_{t=1}^{T}(y_t - S_t^*)^2 \right.$$
$$\left. + \lambda\Sigma_{t=2}^{T-1}\left[(S_{t+1}^* - S_t^*) - (S_t^* - S_{t-1}^*)\right]^2\right\}$$

前項 $(y_t - S_t^*)^2$ 保持 S^* 與 y 貼近，後項是平滑懲罰項，使趨勢變得平滑。

```
S = solve(I + penalty, y)
print(S)
```

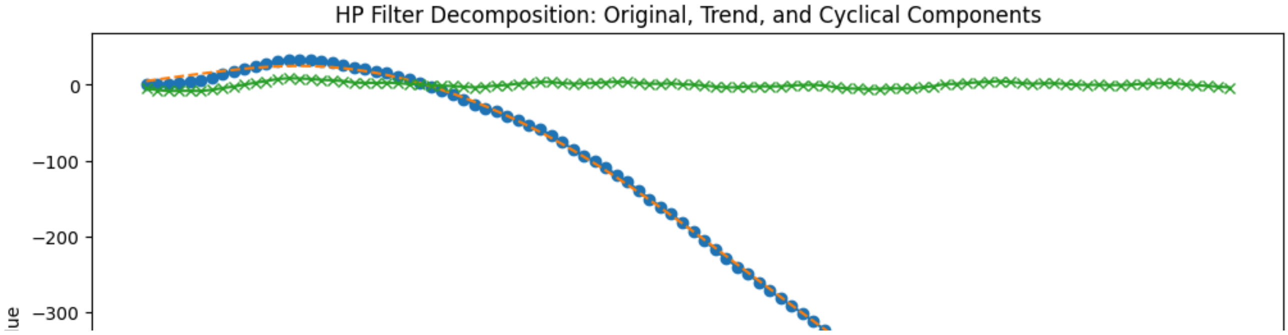
```
[   4.96872517    6.92431935    8.87680806   10.81875818   12.73759333
   14.61573696   16.43062334   18.15458781   19.75578346   21.19963433
   22.45007193   23.47120916   24.22884553   24.69182456   24.83370711
   24.63343312   24.07510586   23.14763872   21.84399266   20.16084082
   18.09790074   15.65662108   12.84002168    9.65262907    6.10068866
    2.19171829   -2.06605068   -6.66393975  -11.5935911   -16.84728796
  -22.41844304  -28.30206682  -34.49541335  -40.99727323  -47.80708133
  -54.92449284  -62.34825274  -70.07563602  -78.10156589  -86.41876711
  -95.01856269 -103.89136565 -113.02653253 -122.4119392  -132.03346499
 -141.87450853 -151.91628021 -162.13840724 -172.51970122 -183.03815224
 -193.67060879 -204.39347379 -215.18312687 -226.01653077 -236.8722191
```

```
-247.7308745  -258.57524838 -269.38949687 -280.15905019 -290.87069402
-301.51247189 -312.07301393 -322.54120886 -332.90604275 -343.15718693
-353.28639822 -363.28848225 -373.16148548 -382.9070116  -392.52999137
-402.03830622 -411.44289154 -420.75768579 -429.99868321 -439.18306498
-448.32739864 -457.44751021 -466.55786769 -475.67092526 -484.79666044
-493.94205227 -503.11179376 -512.30912934 -521.5364539  -530.79497095
-540.08465588 -549.40470508 -558.75427917 -568.13265511 -577.53915443
-586.9733845  -596.43514146 -605.92423517 -615.4396915  -624.97937637
-634.53980807 -644.11619948 -653.70340212 -663.296655   -672.89216372]
```

```
c = y - S
print(c)
```

```
[-4.96872517 -6.92431935 -8.22911953 -8.00035124 -7.98262138 -8.15833694
 -6.69158244 -4.36647131 -2.38806574  0.29024466  2.69855063  4.87042722
  7.54776693  8.60648376  8.26137922  7.6961437   6.47613031  5.93950415
  4.87103284  2.7697636   2.51393136  2.41066241  2.7502414   2.03586541
  1.1416545   0.71539615 -0.51305719 -1.0256924  -1.80720403 -2.5563639
 -3.58977216 -2.45843354 -1.03086938 -0.35250281  1.45635689  2.35197635
  3.7628078   3.5175925   2.24273776  1.45601559  1.69035436  2.36906879
  3.19449886  3.96906503  3.50122832  2.53306515  1.30499136  1.31439514
  1.82658416  0.71289007  0.03728547 -0.93299295 -2.5134053  -3.43839054
 -3.31009182 -2.24754592 -2.03849906 -2.16878996 -2.01251261 -0.93859961
 -0.41372681 -0.15574883 -1.09645292 -3.33672468 -4.87816032 -5.18528884
 -5.69155472 -5.32336851 -4.72102339 -4.88634437 -4.80493467 -3.28921793
 -1.89911829  0.98182817  1.18641387  2.17285396  3.22211904  3.96262268
  4.79758718  3.65766038  2.31771834  1.35923853  1.90624685  1.96497394
  1.24639992  0.05723673 -0.18616007 -0.07128088 -0.45736003 -0.30204837
 -0.02192841  1.25436342  1.85593892  2.15621489  2.08861124  0.57823947
 -0.6200523  -1.54647758 -2.46173915 -3.60933202]
```

```
plt.figure(figsize=(12, 6))
plt.plot(y, label="Original Data (y)", marker="o")
plt.plot(S, label="Trend Component ($S^*$)", linestyle="--")
plt.plot(c, label="Cyclical Component (c)", marker="x")
plt.legend()
plt.title("HP Filter Decomposition: Original, Trend, and Cyclical Components")
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()
```



HPF 應用

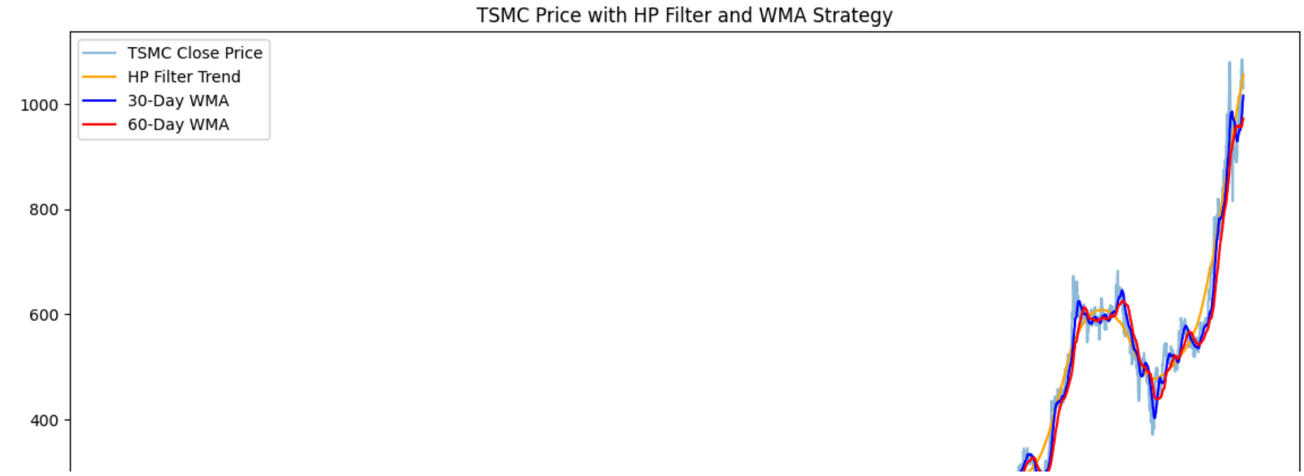
```
# 下載台積電的歷史資料 (2330.TW)
ticker = '2330.TW'
```

```
start_date = '2000-01-01'
end_date = '2024-10-31'

# 計算短期與長期加權移動平均（30 日和 60 日）
short_window = 30
long_window = 60
```

以年頻率為基準

```
data = yf.download(ticker, start=start_date, end=end_date)

# 提取收盤價並使用 HP 濾波器來去除短期波動
close_prices = data['Close']

# 以年頻率為基準(Hodrick & Prescott,1980；Backus & Kehoe, 1992)，λ 值之訂定方式為資料頻率相對一年的平方乘以100，
cycle, trend = hpfilter(close_prices, lamb=365**2 * 100)

# 計算短期與長期加權移動平均（30 日和 60 日）
short_window = 30
long_window = 60
data['Short_WMA'] = close_prices.rolling(window=short_window).mean()
data['Long_WMA'] = close_prices.rolling(window=long_window).mean()

# 交易策略：當短期 WMA 上穿長期 WMA 為買入訊號，下穿為賣出訊號
data['Signal'] = 0
data.loc[data.index[short_window:], 'Signal'] = np.where(
    data['Short_WMA'][short_window:] > data['Long_WMA'][short_window:], 1, -1
)
data['Position'] = data['Signal'].shift()

# 計算策略績效
data['Return'] = data['Close'].pct_change()
data['Strategy_Return'] = data['Return'] * data['Position']

# 繪製趨勢、短期與長期移動平均
plt.figure(figsize=(14, 7))
plt.plot(data['Close'], label="TSMC Close Price", alpha=0.5)
plt.plot(trend, label="HP Filter Trend", color='orange')
plt.plot(data['Short_WMA'], label="30-Day WMA", color='blue')
plt.plot(data['Long_WMA'], label="60-Day WMA", color='red')
plt.title("TSMC Price with HP Filter and WMA Strategy")
plt.legend()
plt.show()

# 計算績效指標
annual_return = np.prod(1 + data['Strategy_Return'].dropna())**(252 / len(data.dropna())) - 1
annual_volatility = data['Strategy_Return'].std() * np.sqrt(252)
sharpe_ratio = annual_return / annual_volatility

print("年化報酬率:", annual_return)
print("年化波動率:", annual_volatility)
print("夏普比率:", sharpe_ratio)
```

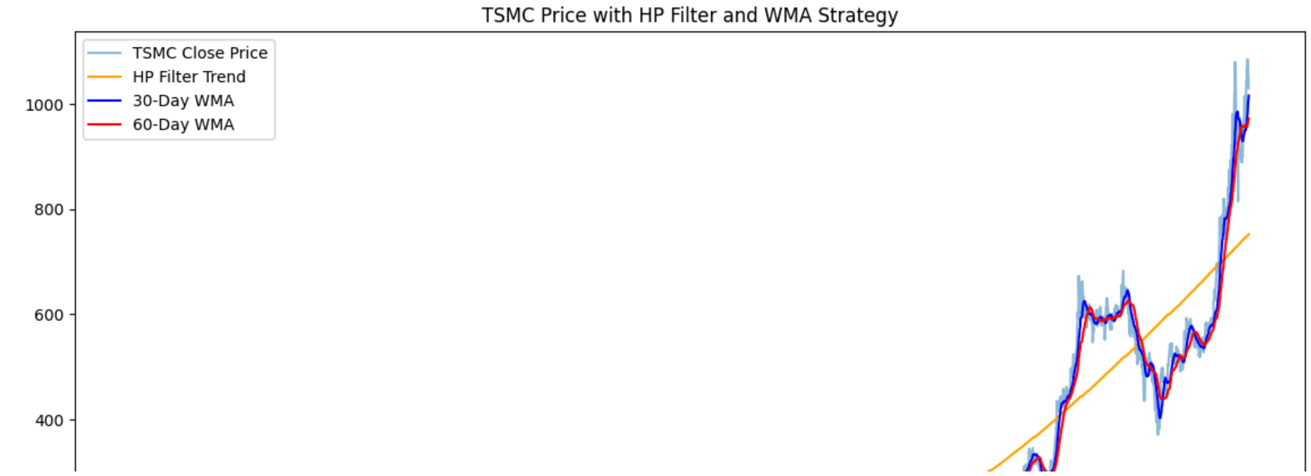
```
[*********************100%***********************]  1 of 1 completed
```



以季頻率為基準

```
data = yf.download(ticker, start=start_date, end=end_date)

# 提取收盤價並使用 HP 濾波器來去除短期波動
close_prices = data['Close']

# 以季為基準(Ravn & Uhlig, 2002)，訂定方式為資料頻率相對一季的四次方乘以 1600，例如日頻率為一季相當 90 日，
cycle, trend = hpfilter(close_prices, lamb=90**4 * 1600)

data['Short_WMA'] = close_prices.rolling(window=short_window).mean()
data['Long_WMA'] = close_prices.rolling(window=long_window).mean()

# 交易策略：當短期 WMA 上穿長期 WMA 為買入訊號，下穿為賣出訊號
data['Signal'] = 0
data.loc[data.index[short_window:], 'Signal'] = np.where(
    data['Short_WMA'][short_window:] > data['Long_WMA'][short_window:], 1, -1
)
data['Position'] = data['Signal'].shift()

# 計算策略績效
data['Return'] = data['Close'].pct_change()
data['Strategy_Return'] = data['Return'] * data['Position']

# 繪製趨勢、短期與長期移動平均
plt.figure(figsize=(14, 7))
plt.plot(data['Close'], label="TSMC Close Price", alpha=0.5)
plt.plot(trend, label="HP Filter Trend", color='orange')
plt.plot(data['Short_WMA'], label="30-Day WMA", color='blue')
plt.plot(data['Long_WMA'], label="60-Day WMA", color='red')
plt.title("TSMC Price with HP Filter and WMA Strategy")
plt.legend()
plt.show()

# 計算績效指標
annual_return = np.prod(1 + data['Strategy_Return'].dropna())**(252 / len(data.dropna())) - 1
annual_volatility = data['Strategy_Return'].std() * np.sqrt(252)
sharpe_ratio = annual_return / annual_volatility

print("年化報酬率:", annual_return)
print("年化波動率:", annual_volatility)
print("夏普比率:", sharpe_ratio)
```

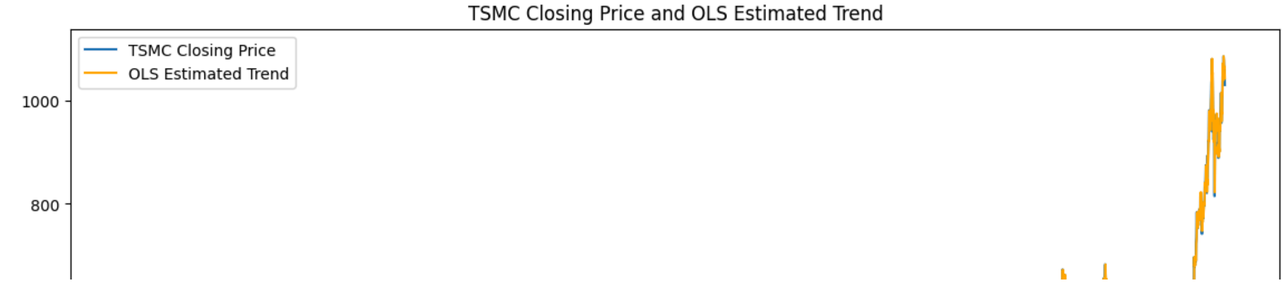
```
[*********************100%***********************]  1 of 1 completed
```



OLS 迴歸法

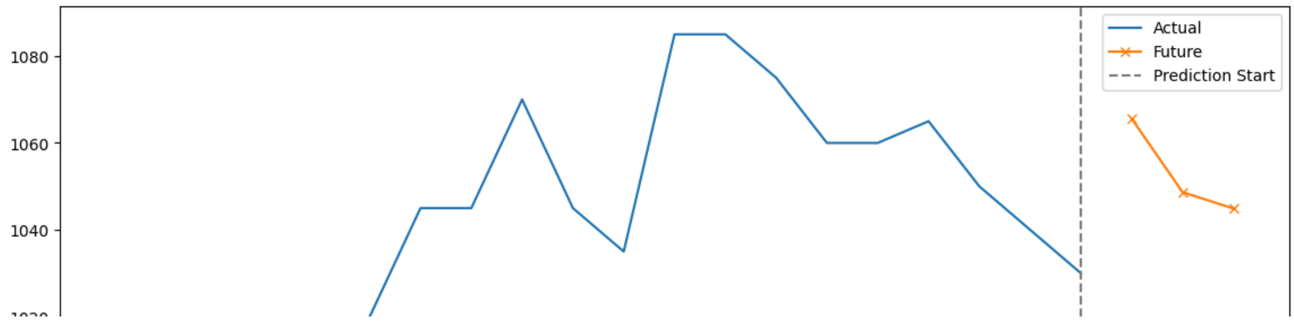
```
data_raw = yf.download(ticker, start=start_date, end=end_date)
data = data_raw["Close"].dropna().reset_index(drop=True)[ticker]

# 定義滯後變量
data_lagged = pd.DataFrame({
    'y_t': data,
    'y_t-1': data.shift(1),
    'y_t-2': data.shift(2),
    'y_t-3': data.shift(3),
    'y_t-4': data.shift(4),
})
data_lagged = data_lagged.dropna()

# 使用OLS回歸模型來估計趨勢
X = data_lagged[['y_t-1', 'y_t-2', 'y_t-3', 'y_t-4']]
X = sm.add_constant(X)  # 添加常數項
y = data_lagged['y_t']

model = sm.OLS(y, X)
results = model.fit()

# 生成趨勢預測
data_lagged['Trend'] = results.predict(X)

# 繪製股價與趨勢線
plt.figure(figsize=(14, 7))
plt.plot(data, label="TSMC Closing Price")
plt.plot(data_lagged['Trend'], label="OLS Estimated Trend", color="orange")
plt.title("TSMC Closing Price and OLS Estimated Trend")
plt.xlabel("Date")
plt.ylabel("Price")
plt.legend()
plt.show()

mse = mean_squared_error(data_raw["Close"].dropna()[4:], data_lagged['Trend'])
rmse = np.sqrt(mse)
rmse
```

```
[*********************100%***********************]  1 of 1 completed
```




```
future_predictions = []
last_known_values = y[-4:]  # 使用最近的四個數據點作為起點進行預測

n = 3
for _ in range(n):  # 預測未來 N 個點
    # 構建當前的滯後值，並進行預測
    X_pred = [1] + list(last_known_values)  # 添加常數項 1
    y_next = round(results.predict([X_pred])[0], 2)  # 預測下一點
    future_predictions.append(y_next)  # 保存結果

    # 更新滯後變量
    last_known_values = np.roll(last_known_values, -1)  # 將滯後數據往前移動
    last_known_values[-1] = y_next  # 更新最新預測值

display(future_predictions)

# 繪圖
recent_points = y[-20:]
plt.figure(figsize=(14, 7))
plt.plot(range(len(recent_points)), recent_points, label='Actual')
plt.plot(range(len(recent_points), len(recent_points) + len(future_predictions)), future_predictions, label='Future', marker=
plt.axvline(len(recent_points) - 1, color='gray', linestyle='--', label='Prediction Start')
plt.legend()
plt.show()
```

```
[1065.56, 1048.67, 1044.9]
```



參考文獻

[Hodrick-Prescott濾波器的參數調整對移動平均技術分析之績效影響－以日頻外匯價格為例]
[WHY YOU SHOULD NEVER USE THE HODRICK-PRESCOTT FILTER]