偵測時間序列中的 baseline 偏移是一個常見的問題，尤其在信號處理和統計分析中。考慮到已經嘗試了多種方法，這邊會提供幾個額外的方法來偵測 baseline 偏移，並且對一些已經使用的方法提出改進建議。

1. 移動平均與標準差法 (Moving Average and Standard Deviation)

這是一種簡單且有效的方法，用來偵測時間序列的局部變化。

方法步驟：
計算序列的移動平均 (Moving Average) 和移動標準差 (Moving Standard Deviation)。
將序列中每一點減去其對應的移動平均值。
如果差值超過某個標準差的多倍數 (例如 3 倍標準差)，則認為存在偏移。
這個方法能夠有效平滑掉局部的震盪，同時考慮到變化的趨勢。

2. 變點檢測 (Change Point Detection)

變點檢測算法專門用來偵測序列中統計特性的變化。

常見的變點檢測算法：
CUSUM (Cumulative Sum Control Chart)：適合偵測序列中的小幅偏移。
BOCPD (Bayesian Online Change Point Detection)：適合實時偵測序列中的變化。
Pelt (Pruned Exact Linear Time)：這是一種快速且精確的變點檢測方法。
(1)以CUSUM舉例：
CUSUM 是累積和控制圖，用於監測數據偏移。
計算 CUSUM 累積和序列：

$$C_t = \max(0, C_{t-1} + (x_t - (\mu + k\sigma)))$$

其中 k是靈敏度參數。當 Ct 超過預設閾值 h時，判斷為變點。

```
import numpy as np

def cusum(data, target_mean, target_std, k, h):
    n = len(data)
    cusum_pos = np.zeros(n)
    cusum_neg = np.zeros(n)

    for i in range(1, n):
        cusum_pos[i] = max(0, cusum_pos[i-1] + (data[i] - (target_mean + k * target_std)))
        cusum_neg[i] = max(0, cusum_neg[i-1] - (data[i] - (target_mean - k * target_std)))

        if cusum_pos[i] > h or cusum_neg[i] > h:
            return i, data[i]

    return None, None

# Example usage
data = np.random.normal(0, 1, 100)
data[50:] += 3  # Introduce a shift
index, value = cusum(data, target_mean=0, target_std=1, k=0.5, h=5)
print(f"Change point detected at index {index}, value {value}")
```

(2)以Pruned Exact Linear Time舉例:
方法步驟:
目標函數:定義變點問題的目標函數，即數據的分段加總成本(通常是負對數似然)。
動態規劃:使用動態規劃方法來最小化目標函數。
剪枝策略:使用剪枝策略以提高算法的效率。

```
pip install ruptures
```

```
import numpy as np
import ruptures as rpt
import matplotlib.pyplot as plt

# 生成示例數據
n_samples, n_dims = 500, 1
n_bkps = 3  # 3個變點
signal, bkps = rpt.pw_constant(n_samples, n_dims, n_bkps, noise_std=1)

# Pelt算法的變點檢測
model = "l2"  # 成本函數("l2"表示歐幾里得距離)
algo = rpt.Pelt(model=model).fit(signal)
result = algo.predict(pen=10)  # 'pen'是懲罰參數，用於控制變點數量

# 視覺化結果
rpt.display(signal, bkps, result)
plt.show()

# 打印變點位置
print("True breakpoints:", bkps)
print("Detected breakpoints:", result)
```

【詳細解釋】

數據生成:使用 ruptures 庫的 pw_constant 函數生成含有變點的時間序列數據。

Pelt 算法:使用 rpt.Pelt 來實現 Pelt 算法，並設置成本函數為 "l2"(歐幾里得距離)。

變點檢測:使用 algo.predict(pen=10) 進行變點檢測，其中 pen 是懲罰參數，用於控制變點數量。

結果視覺化:使用 rpt.display 將原始數據、真實變點和檢測到的變點可視化。

打印結果:顯示真實變點和檢測到的變點位置。

【參數調整】

懲罰參數 (pen):調整該參數以控制檢測到的變點數量。較大的值會減少變點數量，較小的值會增加變點數量。

成本函數 (model):根據數據特性選擇合適的成本函數。常見的選擇包括 "l2"(歐幾里得距離)和 "rbf"(徑向基函數)。

3. 時間序列分段法 (Time Series Segmentation)

將時間序列分段，並比較不同段之間的統計特性。

方法步驟:
使用聚類算法(例如 K-means)或基於動態規劃的分段方法(例如 Bottom-Up segmentation)將時間序列分段。
比較不同段的平均值或中位數，確定是否存在明顯的偏移。

```
from sklearn.cluster import KMeans
import numpy as np

def segment_time_series(data, n_segments):
    n = len(data)
    X = np.arange(n).reshape(-1, 1)

    kmeans = KMeans(n_clusters=n_segments)
    kmeans.fit(X)

    segments = np.split(data, np.where(np.diff(kmeans.labels_))[0] + 1)
    return segments

# Example usage
data = np.random.normal(0, 1, 100)
data[50:] += 3  # Introduce a shift
segments = segment_time_series(data, 2)
for i, segment in enumerate(segments):
    print(f"Segment {i}: mean = {np.mean(segment)}, std = {np.std(segment)}")
```

4. 高斯混合模型 (Gaussian Mixture Model, GMM)

使用 GMM 來建模時間序列，並分析模型中的成分來判斷基線偏移。

方法步驟：

使用 Expectation-Maximization (EM) 算法來估計 GMM 的參數。

分析各個高斯成分的均值和方差，找出基線偏移。

```
from sklearn.mixture import GaussianMixture
import numpy as np

def fit_gmm(data, n_components):
    gmm = GaussianMixture(n_components=n_components)
    gmm.fit(data.reshape(-1, 1))
    return gmm

# Example usage
data = np.random.normal(0, 1, 100)
data[50:] += 3  # Introduce a shift
gmm = fit_gmm(data, 2)
print("Means:", gmm.means_)
print("Variances:", gmm.covariances_)
```

5. 使用時間序列特徵進行分類 (Classification with Time Series Features)

將時間序列特徵提取後作為分類問題來處理。

方法步驟：

提取時間序列的特徵（例如平均值、標準差、峰度、偏度等）。

使用機器學習分類算法（例如隨機森林、支持向量機）來判斷序列是否存在基線偏移。

```
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import kurtosis, skew
import numpy as np

def extract_features(data):
    return [np.mean(data), np.std(data), skew(data), kurtosis(data)]

def classify_segments(data, labels):
    features = np.array([extract_features(segment) for segment in data])
    clf = RandomForestClassifier()
    clf.fit(features, labels)
    return clf

# Example usage
data = [np.random.normal(0, 1, 50), np.random.normal(3, 1, 50)]
labels = [0, 1]  # 0 for no shift, 1 for shift
clf = classify_segments(data, labels)

# Predict new data
new_data = np.random.normal(0, 1, 50)
new_features = extract_features(new_data)
prediction = clf.predict([new_features])
print("Prediction:", prediction)
```

6. 使用自相關函數 (Autocorrelation Function, ACF)

自相關函數能夠揭示序列中的周期性特徵及其偏移。

方法步驟：
計算序列的自相關函數。
分析自相關函數的變化，確定是否存在持續的偏移。

```
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf

def analyze_acf(data):
    plot_acf(data)
    plt.show()

# Example usage
data = np.random.normal(0, 1, 100)
data[50:] += 3  # Introduce a shift
analyze_acf(data)
```

總結

針對需求，推薦從變點檢測 (如 CUSUM 或 BOCPD) 開始，因為這些方法特別適合偵測時間序列中的統計特性變化。移動平均與標準差法也值得嘗試，因為它能夠平滑局部震盪，提供穩定的偏移檢測。

希望這些方法能夠幫助你更準確地偵測時間序列中的 baseline 偏移。如果有需要更詳細的算法實現或數學推導，請隨時告訴我。

【相關文獻】
https://www.newton.ac.uk/files/poster/icpw01-icpmai01-226532.pdf

[PELT]

Pelt (Pruned Exact Linear Time) 是一種高效的變點檢測算法，基於動態規劃和剪枝策略。以下是 Pelt 算法的數學表示及其原理的詳細解釋。

Pelt 算法概述

Pelt 算法的目的是在時間序列中找到變點，使得每個段的統計特性（例如平均值、方差）盡可能均勻。這可以形式化為最小化總成本函數的問題。

數學表示

成本函數 (Cost Function)

假設給定一個時間序列 $\{xt\}$, where t=1,2 … T, 我們希望找到變點 $\{ti\}$使得每段的總成本最小。每段的成本通常是段內數據的某種不均勻度測度，例如歐幾里得距離平方和。定義段s到e的成本為 $C(s,e)$。總成本函數為：

$$\sum_{i=0}^{K} C(t_i, t_{i+1})$$

其中 $t_0 = 1$ 和 $t_{K+1} = T$。

動態規劃 (Dynamic Programming)

使用動態規劃來最小化總成本。設 $F(t)$為從 1 到 t 的最小成本，則有遞推關係：

$$F(t) = \min_{s<t} \{F(s) + C(s,t) + \beta\}$$

其中 β 是懲罰項，控制變點的數量。

剪枝策略 (Pruning)

為了提高計算效率，Pelt 使用了一種剪枝策略，即在計算 $F(t)$時，只考慮那些有可能導致最小成本的候選變點 s。

Pelt 算法的數學原理

<u>定理</u>：Pelt 算法的最優性－Pelt 保證找到全局最優的變點集，即找到使總成本函數最小的變點集。

<u>性質</u>：時間複雜度－Pelt 算法的時間複雜度為 $O(T)$，這是通過剪枝策略實現的。

<u>數學細節</u>：

定義和性質

1. 成本函數 $C(s,t)$：

   - 定義：段 s 到 t 的成本通常是段內數據的不均勻度測度，例如：

$$C(s,t) = \sum_{i=s}^{t} (x_i - \mu_{s:t})^2$$

     其中 $\mu_{s:t}$ 是段內的平均值。

2. 懲罰項 β：

   - 定義：懲罰項控制變點的數量，值越大，檢測到的變點越少。

3. 動態規劃狀態轉移方程 $F(t)$：

   - 定義：從 1 到 t 的最小成本，狀態轉移方程為：

$$F(t) = \min_{s<t} \{F(s) + C(s,t) + \beta\}$$

4. 剪枝策略：

   - 性質：在計算 $F(t)$ 時，只考慮那些有可能導致最小成本的候選變點 s，避免了不必要的計算。

```
import numpy as np

def calculate_cost(data, s, t):
    segment = data[s:t+1]
    mean_segment = np.mean(segment)
    cost = np.sum((segment - mean_segment) ** 2)
    return cost

def pelt(data, penalty):
    n = len(data)
    F = np.zeros(n+1)  # 累計成本
    F.fill(np.inf)
    F[0] = 0

    # 用於存儲變點的位置
    cps = {0: []}

    for t in range(1, n+1):
        for s in range(t):
            cost = calculate_cost(data, s, t-1)
            if F[s] + cost + penalty < F[t]:
                F[t] = F[s] + cost + penalty
                cps[t] = cps[s] + [t]
```

```
    return cps[n], F[n]

# 示例數據
data = np.random.normal(0, 1, 100)
data[50:] += 3  # 引入變點
penalty = 10

# 執行 Pelt 算法
change_points, total_cost = pelt(data, penalty)

# 打印變點位置
print("Detected change points:", change_points[:-1])  # 最後一個點是數據的結尾
print("Total cost:", total_cost)

def calculate_cost(data, s, t):
    segment = data[s:t+1]
    mean_segment = np.mean(segment)
    cost = np.sum((segment - mean_segment) ** 2)
    return cost

def pelt(data, penalty):
    n = len(data)
    F = np.zeros(n+1)  # 累計成本
    F.fill(np.inf)
    F[0] = 0

    # 用於存儲變點的位置
    cps = {0: []}

    for t in range(1, n+1):
        for s in range(t):
            cost = calculate_cost(data, s, t-1)
            if F[s] + cost + penalty < F[t]:
                F[t] = F[s] + cost + penalty
                cps[t] = cps[s] + [t]

    return cps[n], F[n]
```

```
data = np.random.normal(0, 1, 100)
data[50:] += 3  # 引入變點
penalty = 10

# 執行 Pelt 算法
change_points, total_cost = pelt(data, penalty)

# 打印變點位置
print("Detected change points:", change_points[:-1])  # 最後一個點是數據的結尾
print("Total cost:", total_cost)
```