Détection d'une rupture en ligne - méthode FOCuS

Projet d'Algorithmique (M2 Data Science 2022/2023)

Data Saiyentist

10 février 2023

Contents

Configuration de l'espace de travail

La librairie de notre travail est disponible sur Github :

```
# devtools::install_github("DataSaiyentist/projetAlgo")
library(projetAlgo)
```

Description du problème et objectif

La détection de rupture est un domaine de recherche crucial dans de nombreux secteurs qui requièrent une surveillance en temps réel des données. Cela peut inclure des anomalies dans des données médicales, des tendances de trafic sur un site Web et bien plus encore.

Plus précisément, nous avons une série temporelle $(X_t)_{t\in[1,T]}$ qui mesure une certaine quantité dans le temps comme celle-ci (générée aléatoirement) :

```
data <- c(rnorm(50, 0, 1), rnorm(100, 10, 1), rnorm(25, 5, 1), rnorm(100, 0, 2))
break_plot(data)
```

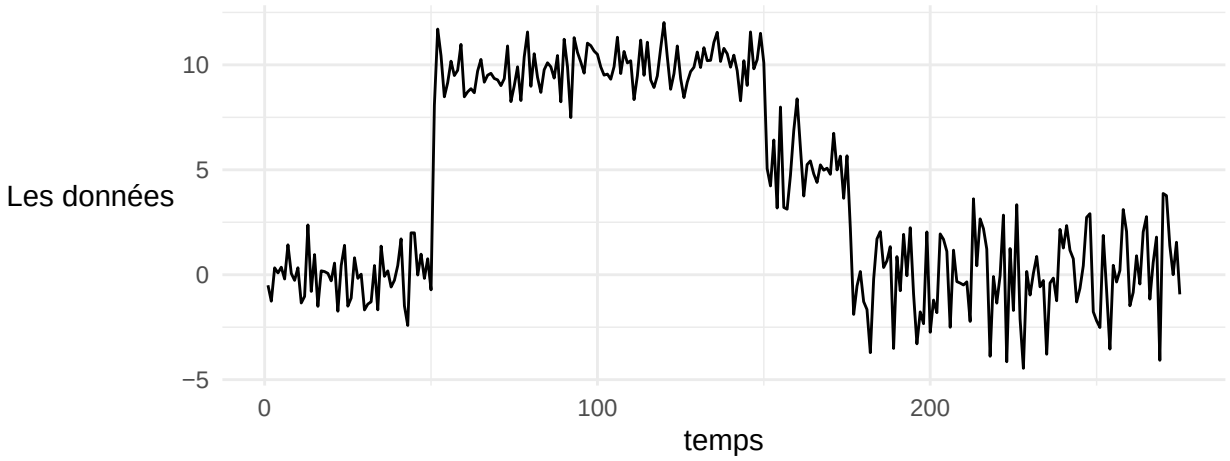

Figure 1: Données générées aléatoirement

Et nous cherchons à identifier si, et si oui quand, le comportement de la série change de manière significative. Par exemple, en surveillant les données de trafic sur un site Web, nous pouvons détecter une rupture dans le nombre de visiteurs qui peut être causée par un bug technique ou une mise à jour.
Comme on peut le voir dans cet exemple, la capacité à détecter les ruptures est extrêmement importante, parce que cela permet notamment de prendre des décisions rapides et informées.

Bien que les ruptures peuvent être faciles à identifier pour un humain, elles peuvent être au contraire difficiles à détecter pour une machine, car elles peuvent se produire à n'importe quel moment et ne sont pas toujours évidentes comme dans cet exemple plus complexe :

```
data2 <- c(rnorm(25, 3, 1), sample(1:5, 50, replace = TRUE), rnorm(50, 3, 1))
par(mar = c(0, 0, 0, 0)+.2); plot(data2)
abline(v = 26, col = "red", lty = "dashed"); abline(v = 76, col = "red", lty = "dashed")
```

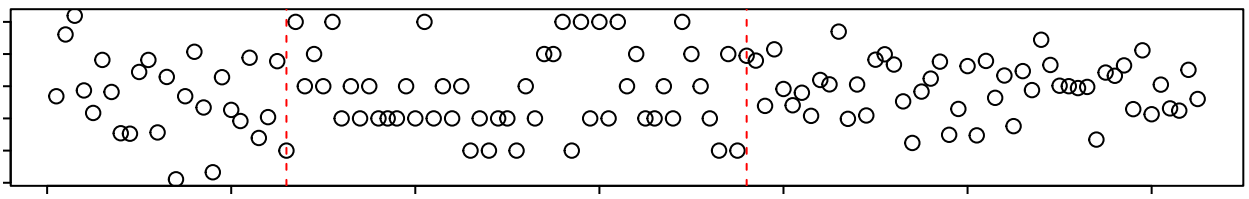

Figure 2: Données générées aléatoirement (avec des entiers au milieu)

Pour répondre au problème, nous utilisons donc des algorithmes de détection de rupture pour automatiser le processus de recherche. Mais dans notre cas, nous nous focaliserons essentiellement sur des algorithmes basés sur des approches statistiques.

Dans ce projet, notre objectif est d'implémenter l'algorithme FOCuS (un algorithme de ruptures en ligne qui s'exécute en temps réel sur les données en cours de collecte) proposé par Romano et al. 2022 en R et C++ et de comparer ses performances (algorithmiques et statistiques) à d'autres algorithmes.

Remarque : Pour faciliter la présentation, et pour rendre les idées concrètes, nous allons considérer, dans ce qui suit, le cas où nous disposons de données provenant de lois normales de variance unitaire.

Les algorithmes naïfs

Test de comparaison

Soient pour tout $t \in [1, T-1]$, $X_1' = (X_1, ..., X_t)$ et $X_2' = (X_{t+1}, ..., X_T)$. On réalise, au choix, les tests de comparaison (avec données non appariées) suivants entre les deux sous-séquences :

- Test de Kolmogorov-Smirnov de comparaison
- Test de la somme des rangs (Wilcoxon)

Puis, les t donnant une p-value plus petite qu'un certain seuil sont considérés comme des instants de saut.

Remarque : Si on souhaite faire la version en ligne, il suffit seulement d'appliquer le test entre les anciennes données et la nouvelle donnée à chaque collecte de données.

On obtiendrait alors les résultats suivants sur les données de la Figure 1 :

```
index_break <- naive_test(data, 1e-20, method = "wilcox")
g1 <- break_plot(data, index_break, "Wilcoxon sur data")

index_break <- naive_test(data, 1e-13, method = "KS")
g2 <- break_plot(data, index_break, "KS sur data"); grid.arrange(g1, g2, nrow = 2)
```

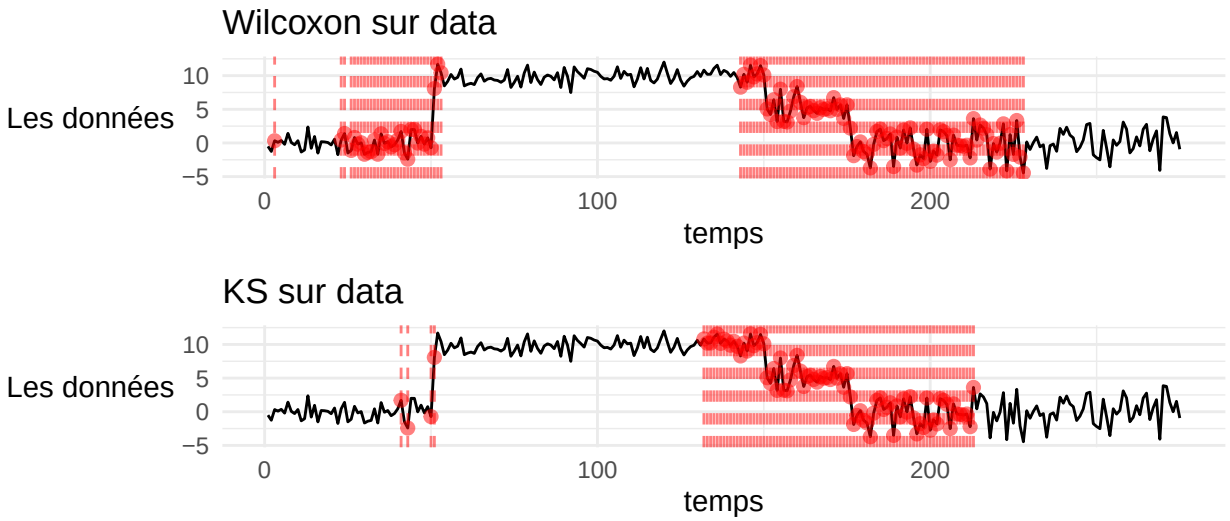

Figure 3: Test de comparaison sur les premières données

Comme on peut le voir sur la Figure 1, cette méthode naïve fait énormément d'erreurs. Malheuresement, il détecte trop de ruptures (malgré un seuil de l'ordre de 10^{-10}). Cette erreur de jugement est probablement due au fait qu'une sous-séquence peut provenir elle-même de plusieurs distributions différentes (par conséquent les tests ne sont plus aussi puissants).

Complexité : Le test de Wilcoxon (version corrigée implémentée à la main) a pour complexité $O(t(T-t))$ pour tout $t \in [1, T-1]$. En effet, sa complexité dépend principalement du temps nécessaire pour comparer terme à terme X_1' et X_2'. Ainsi, avec le test de Wilcoxon, la complexité algorithmique de notre méthode naïve est de l'ordre de $O(T^2)$, parce qu'il y a au total $\sum_{t=1}^{T-1} t(T-t) = \frac{T(T-1)}{2}$ opérations. Quant au test de Kolmogorov-Smirnov (utilisation de la fonction pré-implémenté `ks.test`), il est considérée comme étant de l'ordre de $O(\max\{t, T-t\} log\ \max\{t, T-t\})$. Effectivement, il est nécessaire de trier les données avant de calculer sa statistique de test. Or, le temps nécessaire pour trier des données (de dimension n) est généralement de l'ordre de $O(n log\ n)$. De ce fait, on aurait (grossièrement) :

$$\begin{aligned}
\sum_{t=1}^{T-1} \max\{t, T-t\} log\ \max\{t, T-t\} &= \sum_{t=1, t\geq \frac{T}{2}}^{T-1} \max\{t, T-t\} log\ \max\{t, T-t\} + \sum_{t=1, t<\frac{T}{2}}^{T-1} \max\{t, T-t\} log\ \max\{t, T-t\} \\
&= \sum_{t=1, t\geq \frac{T}{2}}^{T-1} t log\ t + \sum_{t=1, t<\frac{T}{2}}^{T-1} (T-t) log\ (T-t) \\
&\leq 2 \sum_{t=1}^{\frac{T}{2}} t log\ t \quad \text{(par symétrie)} \\
&\leq 2 \sum_{t=1}^{\frac{T}{2}} \frac{T}{2} log\ \frac{T}{2} \quad \text{(par majoration)}
\end{aligned}$$

Donc, avec le test de Kolmogorov-Smirnov, la complexité de la méthode naïve est de l'ordre de $O(T^2 log\ \frac{T}{2})$.

Test de comparaison connaissant le nombre de ruptures

On suppose à présent connaître le nombre de ruptures K dans la série temporelle. De cette manière, il est possible de chercher astucieusement ces points (avec les tests cités précédemment) en segmentant les intervalles de recherche après chaque itération.

Voici un exemple simple (on commmence à la première itération pour mieux visualiser les intervalles de recherche). Supposons qu'on ait trouvé un candidat (d'indice i). Alors :

- Pour tout $t \in [1, i-2]$, on réalise, au choix, les tests de comparaison entre les sous-séquences $X_1' = (X_1, ..., X_t)$ et $X_2' = (X_{t+1}, ..., X_{i-1})$.
- De la même manière, pour tout $t \in [i+1, T-1]$, on réalise, au choix, les tests de comparaison entre les sous-séquences $X_1' = (X_i, ..., X_t)$ et $X_2' = (X_{t+1}, ..., X_T)$.

Remarque : Cette fois-ci, t est un instant de rupture lorsqu'il donne la plus petite p-value.

Avec les données de la Figure 1, on obtient :

```
index_break <- naive_test2(data, K = 10, method = "wilcox")
g1 <- break_plot(data, index_break, "Wilcoxon sur data")

index_break <- naive_test2(data, K = 10, method = "KS")
g2 <- break_plot(data, index_break, "KS sur data"); grid.arrange(g1, g2, nrow = 2)
```

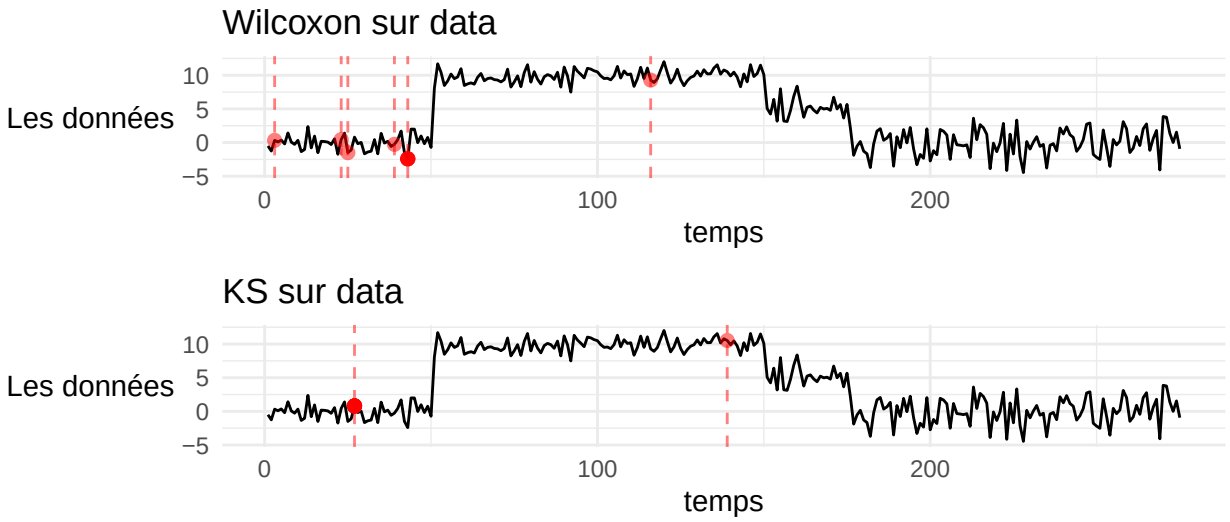

Figure 4: Test de comparaison sachant K sur les premières données

Concernant cette méthode, elle semble faire moins d'erreurs. Mais, elle reste tout de même très loin des résultats attendus. A vrai dire, après-coup, le K se comporte plutôt comme une "limitation de points de rupture à chercher" (cf. Figure 4) par rapport à la première méthode naïve.
De plus, cette méthode est difficile à appliquer dans la vie courante (ne connaissant pas en réalité le nombre de ruptures) et n'est pas applicable pour faire de la détection de ruptures en ligne.

Complexité : Si on reprend les complexités de la section précédente, on sait qu'avec le test de Kolmogorov-Smirnov l'algorithme se comporte en $O(n^2 log\ \frac{n}{2})$, et qu'avec le test de Wilcoxon il se comporte en $O(n^2)$ pour chercher dans une fenêtre de taille n.
Placons-nous dans le pire des cas, ie. les K points de rupture trouvé sont (dans l'ordre) les premiers de la série temporelle. On visualise bien qu'à chaque itération, l'algorithme va chercher sur une fenêtre de taille n, puis $n-1$, etc., jusqu'à $n-K+1$. Donc par majoration, la complexité de l'algorithme est de l'ordre de $O(K \times T^2 log\ \frac{T}{2})$ pour le test de Kolmogorov-Smirnov et de l'ordre de $O(K \times T^2)$ pour le test de Wilcoxon.

Les algorithmes récursifs

CUSUM et Page-CUSUM en ligne

Nous allons à présent introduire des méthodes statistiques plus appropriées pour détecter rapidement une rupture dans une séquence de données. Nous pouvons, d'ores et déjà, affirmer que ces méthodes seront algorithmiquement et statistiquement plus efficaces que les algorithmes naïfs présentés précédemment **(complexité en $O(T)$ pour CUSUM et $O(T^2)$ pour Page-CUSUM)**.

Dans cette section, nous supposerons que la moyenne de pré-rupture μ_0 (ie. la moyenne avant le changemment) est connue. L'objectif est alors de surveiller la valeur absolue des sommes partielles

$$S(s,T) = \sum_{t=s+1}^{T} (x_t - \mu_0)$$

L'idée est que celles-ci devraient être proches de 0 s'il n'y a pas de rupture, et à l'inverse diverger de 0 s'il y a un rupture. Nous allons donc utiliser les statistiques suivantes (introduites par Kirch et al. 2018) :

$$\text{CUSUM} \qquad C(T) = \frac{1}{\sqrt{T}}|S(0,T)|$$
$$\text{Page-CUSUM} \qquad P(T) = \max_{0\leq w<T} \frac{1}{\sqrt{w}}|S(T-w,T)|$$

En testant ces deux algorithmes sur les premières valeurs des données de la Figure 1, nous obtenons :

```
index_break <- CUSUM(data[1:100], threshold = 3, mu0 = 0)$cp
g1 <- break_plot(data[1:100], index_break, "CUSUM sur data[1:100]")

index_break <- pageCUSUM(data[1:100], threshold = 3, mu0 = 0)$cp
g2 <- break_plot(data[1:100], index_break, "Page-CUSUM sur data[1:100]"); grid.arrange(g1, g2, nrow = 2
```

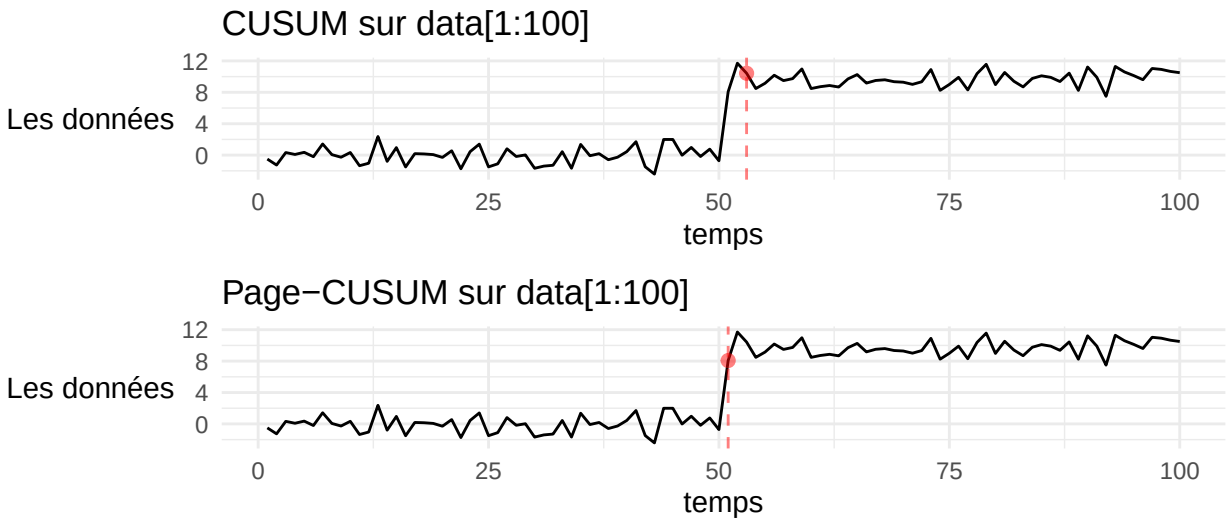

Figure 5: CUSUM et Page-CUSUM sur les 100 premières valeurs

Dans l'exemple précédent, nous observons que ces deux méthodes arrivent à détecter quasi-parfaitement l'instant de rupture. Plus généralement, ces deux compétiteurs sont intéressant algorithmiquement parlant. En effet, ils sont simples à implémenter, parce qu'ils sont récursifs (pour tout s, $S(s,T+1) = S(s,T) + (x_{T+1} - \mu_0)$ et pour tout T, $S(s,T) = S(s+1,T) + (x_s - \mu_0)$).

Complexité : Pour CUSUM, nous calculons des sommes cumulatives (contribution dominante dans la complexité), d'où une complexité de $O(T)$. Et pour Page-CUSUM, il faut rechercher un maximum pour tout $w \in [0,T]$. Or pour tout w, on calcule aussi une somme cumulative (de taille w). Subséquemment, cette recherche nécessite $\sum_{w=0}^{T} w = \frac{T(T+1)}{2}$ opérations, d'où une complexité en $O(T^2)$ au total.

L'inconvénient de ces deux méthodes est qu'elles sont dépendantes de l'hypothèse de connaissance de la moyenne de pré-rupture μ_0 (nous verrons plus loin comment essayer de pallier au problème). De plus, l'approche de Page-CUSUM est un "faux" détecteur de rupture en ligne. En effet, sa complexité algorithmique est de l'ordre de $O(T^2)$. Conséquemment, il est impossible de l'appliquer pour une grande série temporelle (plus de 1000 données par exemple).

Extension : CUSUM et Page-CUSUM hors-ligne

Pour rendre hors-ligne les deux approches précédentes, il suffirait de réappliquer ces procédures lorsqu'on trouve un point de rupture. Mais la question, à présent, est de savoir comment déterminer les moyennes à priori. En fait, on pourrait choisir de manière arbitraire la première valeur de chaque sous-séquences **(pour au final obtenir une complexité algorithmique similaire)**.

Remarque : Ce choix arbitraire peut aussi être appliqué aux méthodes en ligne si besoin.

Voici alors ce que donneraient les versions hors-ligne de ces approches :

```
index_break <- CUSUMs_test(data, threshold = 10, method = "CUSUM")
g1 <- break_plot(data, index_break, "CUSUM hors-ligne sur data")

index_break <- CUSUMs_test(data, threshold = 10, method = "Page-CUSUM")
g2 <- break_plot(data, index_break, "Page-CUSUM hors-ligne sur data"); grid.arrange(g1, g2, nrow = 2)
```

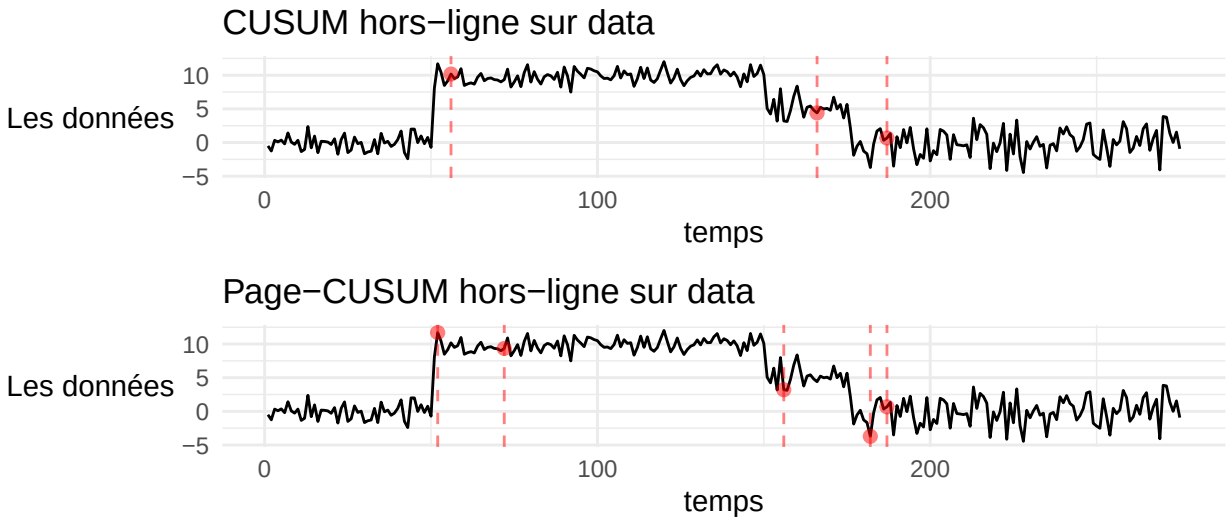

Figure 6: CUSUM et Page-CUSUM hors-ligne sur les premières données

Nous remarquons qu'avec un seuil approprié, les deux méthodes proposées arrivent assez bien à détecter les ruptures (parfois avec un peu de retard ou parfois détecte des fausses ruptures). Mais dans l'ensemble, ces deux méthodes sont fiables en pratique en raison de leurs simplicités.

Complexité : On se place dans le pire des cas, à savoir que ces deux méthodes n'arrivent pas à détecter des ruptures. Dans ce cas, toutes les données seront parcourues et, par extension des versions en ligne, elles ont les mêmes complexités, ie. $O(T)$ pour CUSUM hors-ligne et $O(T^2)$ pour Page-CUSUM hors-ligne.

Méthode séquentielle de Page

Dans cette partie, nous admettons que la moyenne à priori de la série temporelle est nulle, mais que l'on connaît aussi la moyenne à postériori μ_1. Alors, sous l'hypothèse que nos données proviennent de distributions gaussiennes de variance unitaire, la contribution du point x_t dans le rapport des log-vraisemblances est :

$$LR(x_t, \mu_1) = 2\mu_1(x_t - \frac{\mu_1}{2})$$

De cette manière, on peut définir la statistique séquentielle de Page :

$$Q_{T,\mu_1} = \max_{0\leq s\leq T} \sum_{t=s+1}^{T} \frac{1}{2} LR(x_t, \mu_1) \quad \text{avec} \quad \begin{cases} Q_{0,\mu_1} = 0 \\ Q_{T,\mu_1} = \max\{0, Q_{T-1,\mu_1} + \frac{1}{2} LR(x_T, \mu_1)\} \end{cases}$$

Observons donc le comportement de cet algorithme sur les premières valeurs des données de la Figure 1 :

```
index_break <- pageSeq(data[1:100], threshold = 100, mu1 = 10)$cp; break_plot(data[1:100], index_break)
```

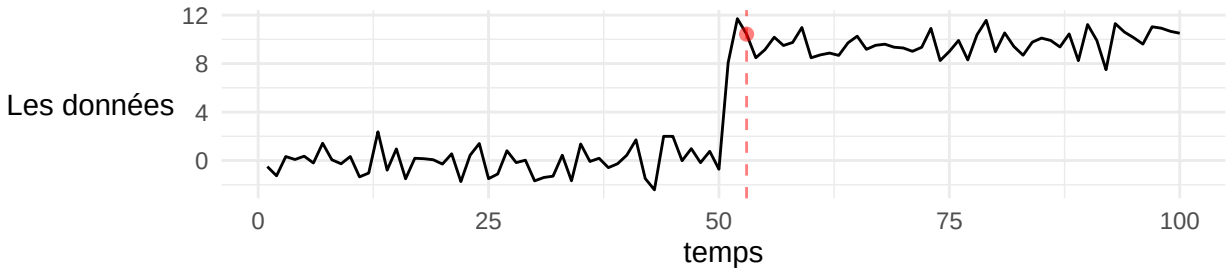

Figure 7: Page séquentielle sur les 100 premières valeurs

Dans cet exemple, nous notons que cette approche en ligne est tout autant satisfaisante (malgré ses hypothèses fortes, à savoir la connaissance de $\mu_0 = 0$ et μ_1), parce qu'elle est aussi récursive (**impliquant une complexité algorithmique de l'ordre de** $O(T)$ et de ce fait une implémentation facile).

Complexité : La contribution majeure, en terme de complexité, est le calcul de la statistique qui est une somme cumulative (dont on force les termes à valoir 0 si ceux-ci sont négatifs à chaque itération). C'est pourquoi sa complexité est de l'ordre de $O(T)$.

FOCuS en ligne

Dans cette partie, nous introduisons l'algorithme clé de notre projet **FOCuS** (en ligne), qui est conçu pour détecter les changements de la moyenne dans les données univariées sous un modèle gaussien. Contrairement à la méthode séquentielle de Page, il peut être utilisé sans forcément connaître la moyenne après le changement. En fait, il met à jour de manière récursive un modèle de quadratique (ie. un polynôme du second degré) par morceaux, et le maximum de ce modèle (sur une grille de plusieurs μ_1) est utilisé comme statistique de test pour détecter un changement.

L'algorithme se décompose en trois parties principales à chaque itération :

- la première partie consiste à construire un modèle de quadratique par morceaux pour plusieurs μ_1.
- la seconde consiste à déterminer la statistique, ie. le maximum sur tous les μ_1.
- la dernière consiste à comparer cette statistique à un certain seuil pour délibérer sur la présence ou non d'un point de rupture.

Plus précisément, l'idée est de reprendre la récursivité séquentielle de Page simultanément pour toutes les valeurs de la moyenne post-changement, mais de dire que Q_{T,μ_1} est un polynôme du second degré par morceaux en μ_1 (cf. Romano et al.2022).

Complexité : A chaque itération, il a été prouvé que le coût de calcul moyen pour construire ce modèle augmente en logarithme de T (cf. Romano et al. 2022), d'où une complexité linéaire au total.