INFO0902: Structures des Données et Algorithmes - Projet 3

Alexandre Andries, Thomas Rotheudt

Table des matières

1 Dynamic Time Warping - Fonction de coût

L'algorithme de *Dynamic Time Warping* est une mesure de similarité entre des séries temporelles. Prenons par exemple deux séries temporelles x_i et x_j, existant dans le même espace dimensionnel. On respecte l'ordre temporel des points.

L'algorithme cherche l'alignement temporel qui minimise la distance absolue moyenne entre les séries alignées. On peut exprimer le problème de manière formelle comme suit (Tavenard, 2021) :

$$DTW_q(x, x') = \min_{\pi \in A(x,x')} \left(\sum_{(i,j)\in\pi} d(x_i, x'_j)^q \right) \tag{1}$$

Où π, un alignement de points de longueur K est une séquence avec K paires d'index telles que $(i_0, j_0), ..., (i_{K-1}, j_{K-1})$ et $A(x, x')$ est l'ensemble de tous les chemins admissibles. Un chemin est considéré comme admissible s'il répond aux conditions suivantes :

1. Le départ et l'arrivée des séries temporelles correspondent ($\pi_0 = (0, 0)$ et $\pi K - 1 = (n - 1, m - 1)$.
2. Les indices i et j de la série augmentent de manière monotone, et chaque indice possible doit apparaître au moins une fois. On note cela mathématiquement comme suit : $i_{k-1} \leq i_k \leq i_{k-1} + 1$ et $j_{k-1} \leq j_k \leq j_{k-1} + 1$.

L'équation (1) correspond à un algorithme d'optimisation sur un ensemble fini, cet ensemble étant l'ensemble des chemins admissibles. Cet ensemble peut devenir très grand et complexe à calculer même pour des séries temporelles de taille raisonnable. Dans le pire cas, pour m et n des dimensions du même ordre, on obtient $O\left(\frac{(3+2\sqrt{2})^n}{\sqrt{n}}\right)$ chemins admissibles différents dans l'ensemble $A(x, x')$, ce qui est exponentiel et dès lors excessivement coûteux à calculer puis comparer, pour en tirer le chemin minimum.

L'objectif de ce projet est de déterminer un algorithme DTW se basant sur la programmation dynamique. On cherche donc à exploiter la récurrence du problème, qui consiste résoudre le problème en le divisant en plus petits sous-problèmes. L'approche se base sur le stockage des sous-problèmes récurrents déjà résolus afin de ne pas gaspiller du temps à les re-résoudre.

Dans le cas du DTW, l'objectif est de résoudre dynamiquement le problème suivant :

$$R_{i,j} = DTW_q(x_{\rightarrow i}, x'_{\rightarrow j})^q \tag{2}$$

Où $x_{\rightarrow i}$ signifie la série temporelle x observée jusqu'à l'indice temporel i inclus.

On obtient que :

$$\begin{aligned} R_{i,j} &= \min_{\pi \in A(x_{\rightarrow i}, x'_{\rightarrow j})} \sum_{(k,l)\in\pi} d(x_k, x'_l)^q && (3) \\ &= d(x_i, x'_j)^q + \min_{\pi \in A(x_{\rightarrow i}, x'_{\rightarrow j})} \sum_{(k,l)\in\pi[:-1]} d(x_k, x'_l)^q && (4) \\ &= d(x_i, x'_j)^q + min(\boldsymbol{R_{i-1,j}}, \boldsymbol{R_{i,j-1}}, \boldsymbol{R_{i-1,j-1}}) && (5) \end{aligned}$$

L'équation (4) et (5) proviennent de la contrainte sur les chemins admissibles et leur correspondances initiale et finale mutuelles.

2 Implémentation de la solution par programmation dynamique

Afin de résoudre le problème par programmation dynamique, nous avons choisi une approche **descendante**. Analysons la complexité en temps et en espace de l'algorithme obtenu (non-optimisé). L'implémentation en langage C de l'algorithme est reprise ci-dessous.

```
1     double dtw(Sketch sketch1, Sketch sketch2, double maxDistance) {
2
3     size_t n = sketch1.size ;
4     size_t m = sketch2.size ;
5     double result ;
6
7     double DTWmtx[n][m] ;
8
9     for(size_t i = 0; i < n ; i++){
10      for(size_t j = 0; j < m; j++){
11          DTWmtx[i][j] = (double)INFINITY ;
12      }
13    }
14    DTWmtx[0][0] = 0;
15
16    for(size_t i = 1; i < n ; i++){
17      for(size_t j = 1 ; j < m ; j++){
18          double dist = d(sketch1.points[i], sketch2.points[j]);
19          DTWmtx[i][j] = dist + min(3, DTWmtx[i-1][j],
20                                       DTWmtx[i][j-1],
21                                       DTWmtx[i-1][j-1]);
22      }
23    }
24
25    return DTWmtx[n-1][m-1] ;
26 }
```

On observe directement que la complexité en espace est $\Theta(m * n)$ car l'algorithme alloue un espace mémoire de dimension $l * m$ afin d'y stocker les résultats des sous-problèmes résolus. La fonction *min* appelée est quant à elle implémentée de manière à avoir une complexité en espace $O(1)$ et n'a donc pas d'incidence sur la complexité spatiale de *dtw*.

On analyse maintenant la complexité en temps de la fonction, en supposant qu'une opération simple correspond à une complexité constante $O(1)$:

$$T(n) = 1 + n * m * 1 + 1 + n * m * 1 + 1 \tag{6}$$
$$= 2 * n * m \tag{7}$$
$$\in \Theta(n * m) \tag{8}$$

Les équations (6) et (7) sont obtenues en regroupant, puis simplifiant, les complexités constantes. La fonction *min* possède quant à elle une complexité constante $O(1)$, car bien qu'elle dépende du nombre d'arguments entrés, dans le cas de la fonction DTW, le nombre d'arguments est toujours équivalent à 3. Elle ne contribue donc qu'à une complexité constante.

L'algorithme DTW possède donc une complexité $\Theta(n * m)$.

3 Complexités en temps et en espace - NearestNeighbours

Analysons les complexités en temps et en espace de la fonction NearestNeighbours pour q le nombre de points du croquis, N le nombre d'échantillons, l la longueur du croquis (constante) et k le nombre de voisins.

3.1 Pire cas

La première boucle de la fonction exécute k tours de boucle, et chaque tours fait appel à la fonciton *bpqInsert* de complexité $O(log(k))$. On a donc $O(k \cdot log(k))$.

La seconde boucle exécute quant à elle $N - k$ tours qui à chaque itération appelle les fonction DTW et *bpqReplaceMaximum*, respectivement de complexité $O(l^2)$ et $O(log(k))$. Cette boucle a donc une complexité $O((N - k) \cdot (l^2 + log(k)))$.

La troisième boucle de la fonction exécute k tours et à chaque tour fait appel à DTW, on en déduit une complexité $O(k \cdot l^2)$.

On obtient au final dans le pire cas que NearestNeighbours $\in O(k \cdot log(k) + (N-k) \cdot (l^2 + log(k)) + k \cdot l^2)$.

3.2 Meilleur cas

L'unique différence entre le pire et le meilleur cas est que dans le meilleur cas, il n'est jamais nécessaire de faire appel à la fonction *bpqReplaceMaximum* dans la seconde boucle. On peut donc écrire que dans le meilleur cas, NearestNeighbours $\in \Omega(k \cdot log(k) + (N - k) \cdot l^2 + k \cdot l^2)$.

4 Taux d'erreur pour $k \in \{1, 3, 7\}$

k =	1	3	7
Taux d'erreur (%)	18.60	18.40	19.80

TABLE 1 – Taux d'erreur pour $k \in \{1, 3, 7\}$ pour la petite base de données.

k =	1	3	7
Taux d'erreur (%)	9.40	10.20	10.20

TABLE 2 – Taux d'erreur pour $k \in \{1, 3, 7\}$ pour la grande base de données.

5 Temps empiriques des calculs de taux d'erreur

k =	1	3	7
Temps de calcul (secondes)	14.551642	16.948275	18.652390

TABLE 3 – Temps de calcul des taux d'erreur pour $k \in \{1, 3, 7\}$ pour la petite base de données.

k =	1	3	7
Temps de calcul (secondes)	120.297928	136.141572	147.328599

TABLE 4 – Temps de calcul des taux d'erreur pour $k \in \{1, 3, 7\}$ pour la grande base de données.

Les résultats expérimentaux corroborent les attentes théoriques relatives à la complexité en temps démontrée formellement. On observe que les temps de calculs augmentent en fonction des k choisis, ainsi que la taille de la base de données analysée.

Bibliographie

[Sta07] Philip Chan Stan Salvador. "FastDTW : Toward Accurate Dynamic Time Warping in Linear Time and Space". In : *Intelligent Data Analysis* 11.5 (2007), p. 561-580.

[al09] Thomas H. Cormen et al. *Introduction to Algorithms*. MIT Press, 2009.

[Zha20] Jeremy Zhang. *Dynamic Time Warping*. 2020. url : `https://towardsdatascience.com/dynamic-time-warping-3933f25fcdd`. (accessed : 22.05.2022).

[Tav21] Romain Tavenard. *An Introduction to Dynamic Time Warping*. 2021. url : `https://rtavenar.github.io/blog/dtw.html`. (accessed : 22.05.2022).

[Wik] Wikipedia. *Dynamic time warping*. url : `https://en.wikipedia.org/wiki/Dynamic_time_warping`. (accessed : 22.05.2022).