**Projet python**

Jeanne DENIEL et Anthony VAÏTILINGOM (Groupe B)

**Sujet 2** : Mesurer les similarités des capteurs pour chaque dimension, qu’en concluez-vous ? Proposez et implémentez un algorithme permettant de mesurer la similarité automatiquement et de la montrer sur les courbes.

1. **Idée de l’algorithme :**

Notre sujet consiste à mesurer la similarité des capteurs pour chaque dimension comme la température ou la luminosité. Directement, nous avons pensé à la notion de distance euclidienne que nous avons vu en mathématiques.

Pour deux vecteurs X et Y dans un espace euclidien de dimension n, avec n ∈ ℕ, on peut noter X et Y comme : X=(x0, x1, …, xn) et Y=(y0, y1, …, yn), alors la distance euclidienne est :

d=√(∑\_(i=0)^n▒〖(xi-yi)²〗)

Le principe de cette distance serait donc de comparer un à un les termes des dimensions pour faire le calcul de distance. Cependant, on remarque qu’il faut que les deux vecteurs (la même dimensions de deux capteurs différents) aient la même taille, soit le même nombre d’éléments, ce qui n’est pas le cas.

Notre premier réflexe a été de vouloir couper le vecteur le plus grand . En effet, nous pensions que les capteurs ne s'allumaient et ne s’éteignaient pas en même temps, ce qui constituerait un point intéressant pour notre étude. Cependant, en regardant la date et surtout l’heure des mesures ne sont pas les mêmes en fonction des capteurs. Les capteurs ne font pas leurs mesures en même temps et pas aux même intervalles, et cela nous empêche d’utiliser la distance euclidienne, elle ignore les dépendances temporelles entre les différentes séries de données, elle ne peut pas comparer la forme des signaux des capteurs.

Comme il était selon nous impossible d’utiliser la distance euclidienne sans couper des valeurs et faire des comparaison hasardeuses, nous avons réfléchi à chercher une méthode qui nous permettrait de comparer deux séries temporelles. L’idée pour nous était de trouver une méthode qui compare les différentes mesures en fonction de la valeur et du temps. Comme les bureaux passent d’un mode « fonctionnement » quand ils sont utilisés, à un mode « repos » quand c’est la nuit et que personne ne travaille, on peut logiquement se dire que lors de la période de repos, les capteurs, même si ils ne prennent pas les mesures exactement au même moment, vont avoir des valeurs de mesures très similaires sur cette période. Ainsi, l’idée serait de comparer, quand la période d’utilisation le permet, les mesures rapprochées en temps entre elles. Cela permettrait aussi de régler le souci de la différence des tailles des dimensions. Une mesure est comparée à la mesure de l’autre capteur qui est la plus logique à comparer, ce n’est pas grave si cette mesure est déjà utilisée dans une autre comparaison.

C’est en cherchant une méthode qui pourrait répondre à nos nombreuses exigences que nous avons découvert le principe de la distance d’alignement temporel dynamique (DTW : Distance Time Warping). La DTW permet de comparer deux séries temporelles de dimensions différentes, le principe de la distance consiste à mettre en correspondance les sous séquences qui "se ressemblent" même si elles ne correspondent pas à un même intervalle de temps.

La comparaison de deux séries temporelles X et Y de dimensions respectives n et m, est basée sur la réplication des valeurs jusqu’à l’obtention de la meilleure correspondance.

1. **Explication de l’algorithme utilisé :**

Voici ci-dessous l’algorithme de Distance Time Warping que nous allons vous détailler ensuite.

Algorithme : Distance Time Warping

1: n← |X|

2: m← |Y|

3: dtw[] ←new [n\*m]

4: dtw(0,0) ←0

5: for i= 1; i ≤ n ; j+ + do

6: dtw(i, 1) ←dtw(i−1,1) + |Xi−Y1|

7: end for

8: for j= 1; j ≤ m ; j+ + do

9: dtw(1, j)←dtw(1, j −1) + |X1−Yj|

10: end for

11: for i= 1; i ≤ n ; j+ + do

12: for j= 1; j ≤ m ; j+ + do

13: dtw(i, j)←|Xi−Yj|+min {dtw(i−1, j); dtw(i, j −1); dtw(i−1, j −1)}

14: end for

15: end for

16: L[] ←new array

17: i=rows(dtw)

18: j=columns(dtw)

19: while (i > 1) & (j > 1) do

20: if i== 1 then

21: j=j−1

22: else if j== 1 then

23: i=i−1

24: else

25: if dtw(i−1, j) == min {dtw(i−1, j); dtw(i, j −1); dtw(i−1, j −1)} then

26: i=i−1

27: else if dtw(i, j−1) == min {dtw(i−1, j); dtw(i, j −1); dtw(i−1, j −1)} then

28: j=j−1

29: else

30: i=i−1; j=j−1

31: end if

32: L.add((i, j))

33: end if

34: end while

35: x[]←new array

36: y[]←new array

37: Distance[]←new array

38: for i= 1; i ≤ len(L) ; do

39: x.append(L(i,0))

40: y.append(L(i,1))

41: for i= 1; i ≤ (len(x)); do

42: Distance.append(np.abs(X(x(i))-Y(y(i))))

43: return (L,Distance)

On commence ligne 3 par créer un tableau de dimensions n et m soit les tailles respectives des deux vecteurs à comparer X et Y. Cette première étape consiste à construire une matrice des coûts locaux d'alignement des vecteurs, dans l’algorithme notée dtw. Le principe est pour chaque case de coordonnées (i,j) de la matrice, i ∈[1 : n], j ∈[1 : m], de calculer la distance euclidienne entre Xi et Yj, plus la distance totale et minimale du trajet pour aller à (i,j). A chaque étape on calcule donc la distance euclidienne entre Xi et Yj qui est la nouvelle étape de la récurrence, plus le trajet le plus court pour y arriver en partant d’une case adjacente et précédente de la case de coordonnées (i,j), les cases adjacentes précédentes étant celles de coordonnées (i-1,j-1), (i-1,j) et (i,j-1).

Chaque case (i,j) est la somme de la distance entre Xi et Yj plus la valeur la plus petite des cases précédentes représentant le trajet avec le moins de distances à emprunter.

On veut en effet un chemin qui permet de parcourir toutes les valeurs des deux vecteurs en prenant le moins de distance possible. Pour cela il faut non seulement comparer les vecteur adéquates entre eux, mais aussi passer par le chemin le plus court et qui va créer le moins de détours illogiques, inutiles et coûteux.

Lignes 5 et 8 on initialise donc la première ligne et la première colonne de la matrice dtw selon le principe de calcul, pour la case de coordonnées (i,j), de la distance euclidienne entre Xi et Yj, et du chemin pour aller des cases précédant (i,j) à (i,j).

Or, pour la première ligne (i=1), on a une seule case existante qui précède (1,j), c’est (1,j-1). Et c’est le même principe pour la première colonne, ce qui explique que l’on ne fait pas de calcul de minimum pour les lignes 6 et 9.

Une fois cela fait, on peut, par récurrence des boucles for, remplir le tableau sur le principe expliqué précédemment et qui est illustré ligne 13.

On a donc, à la sortie des boucles ligne 15, le tableau dtw.

La seconde étape va être de sélectionner le chemin le plus logique de comparaison de X et Y. C'est-à-dire prendre celui pour lequel les comparaisons sont à chaque fois les plus logiques soit les plus petites tout en étant les moins nombreuses possibles , ce qui correspond au chemin parcourant dtw le plus court. Pour cela on fonctionne avec une boucle while dans le sens décroissant de n et m, contrairement à l’initialisation de dtw.

La liste L initialisée ligne 16 va être la liste des coordonnées des points à comparer entre eux avec une distance euclidienne.

Dans la boucle while, le principe, lignes 25, 27 et 29, est, pour la case de coordonnées (i,j), i ∈[1 : n], j ∈[1 : m], de comparer les chemins qui permettent d’accéder à cette case. Ces différents chemins sont soit par dtw(i-1,j), par dtw(i,j-1) ou encore par dtw(i-1,j-1). En regardant laquelle de ces valeurs est la plus petite, en suivant le raisonnement expliqué plus tôt, on peut retrouver laquelle des distances entre les trois valeurs est la plus logique pour arriver à (i,j).

Ligne 32 on ajoute les coordonnées des distances à comparer et à retenir dans l’algorithme.

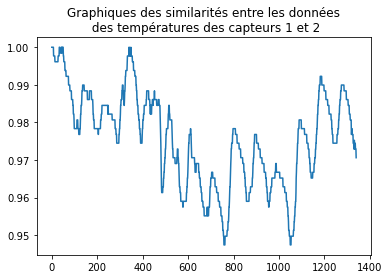
Ligne 35 et 36 on crée deux listes vides afin de ranger par ordre de comparaison les coordonnées à comparer. Pour cela, à l’aide d’une boucle for, ligne 37, on parcourt les tuples de la liste L et on note les coordonnées pour comparer des valeurs respectives de chaque vecteurs X et Y.

Ensuite, ligne 41, dans une autre liste vide nommée Distance, on remplit les valeurs des distances entre les coordonnées que l’on avait trouvé dans la liste L.

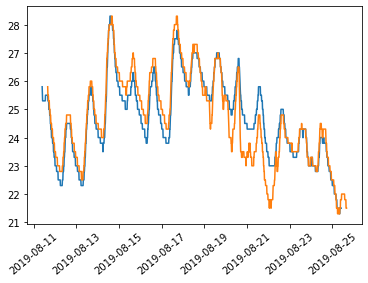
Une fois que nous avons réussi à coder notre algorithme, en le testant nous nous sommes rendus compte d’un autre problème. En effet, quand le sujet parle de similarités, nous avons pensé qu’il fallait renvoyer un pourcentage de "ressemblance". Pour cela, il nous ait venu l’idée de normaliser les distances afin de bien obtenir des pourcentages. C’est en utilisant le procédé de normalisation de Gram-Schmidt que nous avons calculé la norme de la série Distance, afin d’obtenir des pourcentages.

Deuxième petit problème dont nous nous sommes rendus compte, il s’agit de ce que renvoie notre algorithme DTW. En effet il s’agit d’une distance, une sorte de “différence”, or, on veut obtenir des similarités, ce qui s’apparente plus à des ressemblances dans l’idée. C’est pour cela que dans le programme, nous affichons le calcul 1-Norme(DistanceTimeWarping).

Ainsi, voici un exemple de graphe que l’on peut obtenir avec ici les similarités des capteurs :



Ce graphique montre que les courbes sont extrêmement similaires, et que les points comparés deux à deux sont similaires au minimum à 95%. On a sur ce graphique la similiarité en pourcentage de chacune des comparaisons effectuées entre les points de la donnée de la température des capteur 1 et 2.



Le graphique ci-dessus montre l’évolution de la mesure de la température par les capteurs 1 et 2 en fonction du temps. On remarque que les endroits ou les mesures divergent correspondent à des points de similarité bas. Cela nous conforte dans nos résultats obtenus en termes de logique.

1. **Défaut de notre méthode :**

Notre méthode a une complexité temporelle plutôt élevée. En effet, notre programme est de complexité quadratique : O(n²), ce qui, pour le nombre de valeurs que contient le tableau, peut s'avérer long pour un algorithme, et c’est pour cela que notre algorithme peut s’avérer très long à faire tourner.