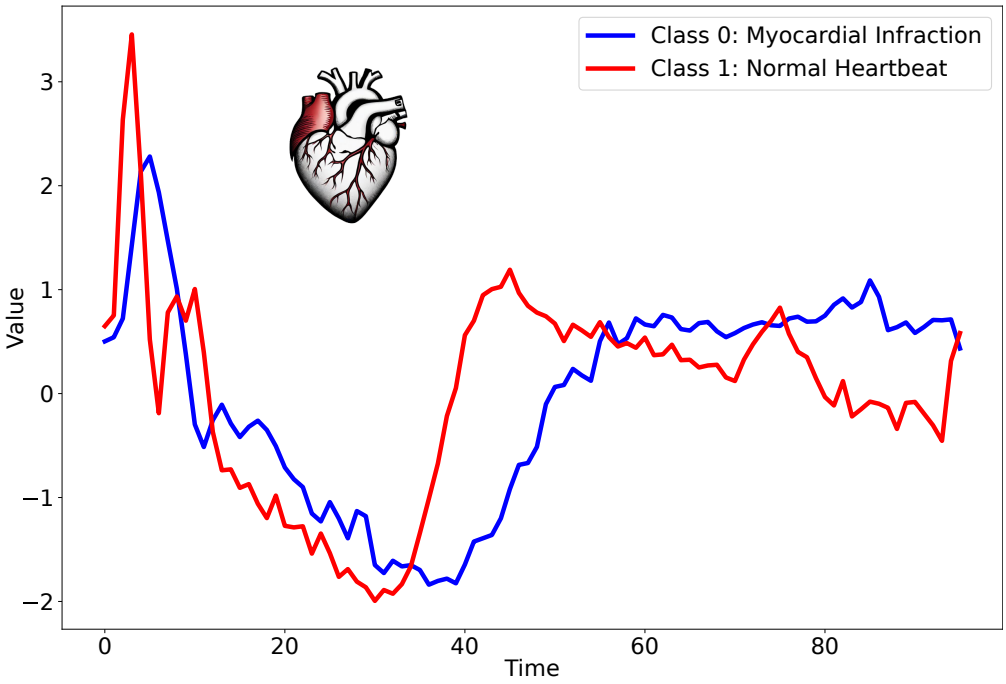
TP: Classification des Séries Temporelles

C++ - ENSISA 2A

Ali El Hadi ISMAIL FAWAZ

November 12, 2024

Les Séries Temporelles



Les séries temporelles sont un type particulier de données qui représentent des séquences de valeurs ordonnées dans le temps. Les données peuvent être horodatées explicitement (une date est associée à chaque valeur) ou implicitement (l'ordre d'occurrence des données est pris en compte).

Ces données peuvent, par exemple, représenter l'évolution des valeurs fournies par un capteur au fil du temps (telles que la température ou la pression), un électrocardiogramme, ou même du son.

La classification des séries temporelles consiste à attribuer une classe à un point de données en fonction de ses caractéristiques. Par exemple, les électrocardiogrammes des patients peuvent être classés comme "normaux" ou "anormaux" en fonction de leurs caractéristiques.

Dans ce contexte, l'algorithme du plus proche voisin est souvent utilisé pour classifier les séries temporelles. Pour une série temporelle à classifier, cet algorithme consiste à trouver la série temporelle "la plus proche" dans un ensemble de séries temporelles dont les classes sont connues. Ainsi, il existe deux ensembles de données : un premier ensemble, appelé "entraînement," composé de séries temporelles avec des classes connues, et un deuxième ensemble, appelé "test," pour lequel nous cherchons à prédire la classe.

Exemple d'un échantillon de série temporelle à partir d'un signal ECG lors de la surveillance du cœur, comme on peut le voir dans la figure ci-dessus. Cet exemple est tiré de l'ensemble de données ECG200 de l'archive UCR [1]. L'ECG200

[1] https://www.cs.ucr.edu/~eamonn/time_series_data_2018/

contient deux ensembles de classes, différenciant si le patient a un problème cardiaque ou non.

Dans cet exercice, votre but est de définir des générateurs de séries temporelles, un classifieur de plus proche voisin (K-Nearest Neighbors: KNN) avec trois mesures de similarité différentes.

Question 1:

Implémentez la classe mère `TimeSeriesGenerator` qui contient un élément : `seed (int)`. Cette classe doit avoir un constructeur par défaut et un constructeur paramétrisé. Les fonctions membres de cette classe sont :

- `vector<double> generateTimeSeries(int)`, une fonction virtuelle pure qui prend comme paramètre la taille d'une série et retourne une série générée dans un vecteur de type `double`.
- `void printTimeSeries(const vector<double>)`, une fonction statique qui imprime la série donnée en paramètres.

Question 2:

Implémentez la classe `GaussianGenerator` qui hérite de la classe `TimeSeriesGenerator` pour générer une série temporelle provenant d'une distribution gaussienne. Utilisez la méthode de Box-Muller pour générer des valeurs gaussiennes.

La classe `GaussianGenerator` contient deux éléments : la moyenne et l'écart type.

Question 3:

Implémentez la classe `StepGenerator` qui hérite de la classe `TimeSeriesGenerator` pour générer une série temporelle provenant d'une fonction de saut $(0, 100)$, c'est-à-dire qu'à chaque instant, une valeur entière entre 0 et 100 est choisie de manière aléatoire avec une probabilité 50% ou bien on garde la valeur de l'instant précédent avec probabilité 50%. On suppose que le premiere instant contient la valeur 0.

Question 4:

Implémentez la classe `SinWaveGenerator` qui hérite de la classe `TimeSeriesGenerator` pour générer des séries temporelles basées sur la fonction sinus. Cette classe a comme éléments : l'amplitude A, la fréquence ω et la phase initiale ϕ, afin d'obtenir la fonction sinus :

$$f(x) = A \cdot \sin(\omega \cdot x + \phi)$$

Question 5:

Implémentez la classe `TimeSeriesDataset` qui représente un jeux de donnée de series temporelles. Les elements de cette classe sont: `znormalize (bool)`, `isTrain (bool)`, `data (vector<double>)`, `labels (vector<int>)`, `maxLength (int)` et `numberOfSamples (int)`.

Znormalization

Pour ne pas travailler avec plusieurs séries de différentes échelles, on normalise chaque série temporelle ajoutée au jeu de données. Normalement, le type de normalisation utilisé pour les séries temporelles est la normalisation Z, par exemple sur une série temporelle de taille L :

$$x = \frac{x - \mu_x}{\sigma_x}$$

μ_x et σ_x sont la moyenne et écart type de la séries x. Si l'élément `znormalize` de la classe `TimeSeriesDataset` est `true`, il faut normaliser chaque séries dans le jeu de donnée.

Mesures de Similarité

Question 6:

Nous devons définir la similarité entre les séries temporelles afin de déterminer ce qu'est un voisin.

Implementez une fonction `double euclidean_distance(const vector<double>, const vector<double>)` qui prend comme argument deux series temporelles et rend la distance Euclideane entres ces 2 series:

$$ED(x, y) = \sqrt{\sum_{i=0}^{len(x)-1}(x[i] - y[i])^2}$$

Cette mesure de similarité suppose un alignement temporel parfait entre les séries temporelles, ce qui n'est pas toujours le cas.

Question 7:

Dynamic Time Warping (DTW)

DTW est une mesure de similarité spécifique aux données de séries temporelles, car elle prend en compte le fait que les deux séries temporelles ne sont pas parfaitement alignées sur l'axe temporel.

$$DTW(x, y) = \min_\pi \sqrt{\sum_{(i,j)\in\pi}(x[i] - y[j])^2}$$

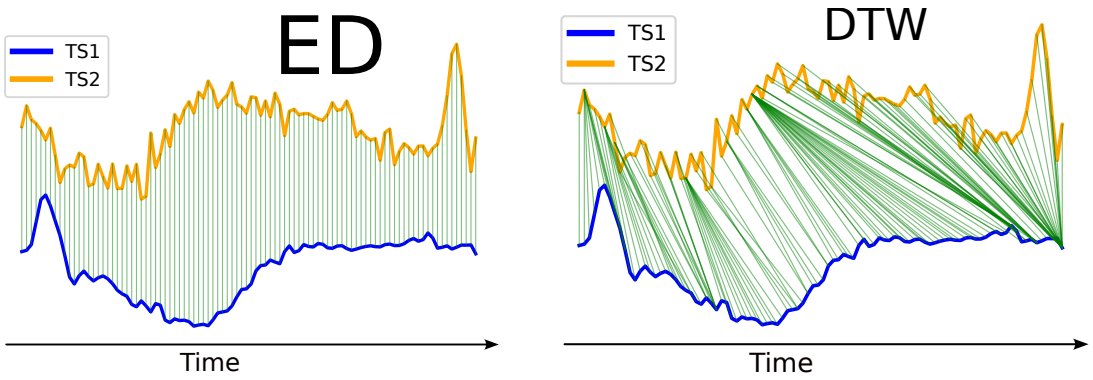
où π est le chemin d'alignement.

Algorithm 1: Dynamic Time Warping (DTW)

Data: Time Series x, Time Series y, both of length L
Result: DTW distance

Initialize a matrix D of size $L+1 \times L+1$;
Set $D[0,0] = 0$;
Set $D[:,0] = \infty$;
Set $D[0,:] = \infty$;
for $i \leftarrow 1$ **to** L **do**
  for $j \leftarrow 1$ **to** L **do**
    Set $d = (x[i] - y[j])^2$;
    Set $D[i][j] = d + \min(D[i, j-1]\,insert,$
      $D[i, j-1]\,delete,$
      $D[i-1, j-1])\ move;$
  end
end
return $\sqrt{D[L, L]}$;

DTW fonctionne en recherchant le chemin d'alignement optimal entre deux séries temporelles en générant d'abord une métrique de différence au carré, voir l'algorithme 1 pour plus d'informations.



Nous pouvons voir dans la figure ci-dessus comment DTW trouve le chemin d'alignement optimal entre les deux séries.

Question 8: Question Bonus

Edit Distance Real Sequences

Lire a propos de cette mesure de similarité et l'implémenter.

Question 9:

Implementez la classe `KNN`, le plus proche voising, les elements de cette classe sont: `k (int)` et `similarity_measure (string)`.

Cette classe doit contenir la fonction `evaluate` qui donne la metrique "accuracy" entre 0 et 1, qui represente le nombre de series correctement classes.

$$accuracy(ytest, ypred) = \frac{1}{len(ytest)} \sum_{i=0}^{len(ytest)-1} (ypred[i] == ytest[i])$$

Dans le cas ou $k > 1$, on choisit la classe la plus populeuse entre les voisins pour la prédiction.

Question 10: Fichier main.cpp

Adapter votre code pour que le fichier main fonctionne. C'est a vous d'ajoutter des fonctions a TimeSeriesDataset et KNN.

```
#include "headers/sin.h"
#include "headers/stp.h"
#include "headers/gau.h"
#include "headers/tsdata.h"
#include "headers/knn.h"
#include<iostream>
#include<vector>

using namespace std;

int main(){

    TimeSeriesDataset trainData(false, true), testData(false, false);

    GaussianGenerator gsg;
    SinWaveGenerator swg;
    StepGenerator stg;

    vector<double> gaussian1 = gsg.generateTimeSeries(11);
    trainData.addTimeSeries(gaussian1, 0);
    vector<double> gaussian2 = gsg.generateTimeSeries(11);
    trainData.addTimeSeries(gaussian2, 0);

    vector<double> sin1 = swg.generateTimeSeries(11);
    trainData.addTimeSeries(sin1, 1);
    vector<double> sin2 = swg.generateTimeSeries(11);
    trainData.addTimeSeries(sin2, 1);

    vector<double> step1 = stg.generateTimeSeries(11);
    trainData.addTimeSeries(step1, 2);
    vector<double> step2 = stg.generateTimeSeries(11);
    trainData.addTimeSeries(step2, 2);

    vector<int> ground_truth;
    testData.addTimeSeries(gsg.generateTimeSeries(11));
    ground_truth.push_back(0);

    testData.addTimeSeries(swg.generateTimeSeries(11));
    ground_truth.push_back(1);

    testData.addTimeSeries(stg.generateTimeSeries(11));
    ground_truth.push_back(2);

    KNN knn_1(1, "dtw");

    cout << knn_1.evaluate(trainData, testData, ground_truth) << endl;

    KNN knn_2(2, "euclidean_distance");

    cout << knn_2.evaluate(trainData, testData, ground_truth) << endl;

    KNN knn_3(3, "euclidean_distance");

    cout << knn_3.evaluate(trainData, testData, ground_truth) << endl;
}
```